Katholieke Universiteit Leuven
Faculteit Toegepaste Wetenschappen
Departement Computerwetenschappen

Patterns and Anti-Patterns in Object-Oriented Analysis

Promotor:
Prof. dr. ir. E. Steegmans

Proefschrift voorgedragen
tot het behalen van het
doctoraat in de toegepaste
wetenschappen door
Frank Devos

Abstract

The major goal of analysis is to establish the functional requirements of the system to be developed and to establish the real-world facts that are involved in these requirements. Because every system is concerned with a certain reality outside itself, analysis of real-world facts is a necessary step in the system development cycle. Analysis is gaining importance as the demand for complex systems grows. It helps developers to understand and document the business domain. Moreover, analysis helps in detecting errors early in the development cycle, thereby reducing the cost to correct them.

In this work, an object-oriented approach was adopted to perform analysis. It is generally accepted that this approach offers significant advantages over more traditional advantages. These advantages are mainly due to proper support for encapsulation, decomposition, specialization and polymorphism. However, we observed that there still exist some serious problems in applying an object-oriented approach during analysis. We believe that the major problems are due to the vague border between analysis and design, to the lack of adequate modelling constructs, guidelines and quality criteria and to problems in transitioning from analysis to design.

The goal of this thesis is threefold. A first objective is to evaluate a part of the Unified Modeling Language (UML), the default standard modelling language for object-orientation, as a language for analysis. A second objective is to offer patterns that guide the analyst in building quality models and allow him to focus on the business domain. A third objective is to define principles that help to evaluate modelling language constructs and guidelines.

We do not consider the UML in its present form as an appropriate modelling language for precisely and completely specifying functional requirements and observed facts in the business domain. Several patterns will be described to represent objects, properties, business rules and functional requirements. Seven principles, such as the principles of completeness, abstraction and extendibility, will be defined to evaluate the proposed modelling constructs and guidelines.

Acknowledgments

"Doctoreren es vele schriv'n"
"Wat niet gemodelleerd kan worden is de waarheid niet"

No thesis is written in a vacuum. During the long process of writing I have received support, without which this thesis would not exist, from colleagues and loved ones.

First of all, I would like to especially thank Prof. Dr. Eric Steegmans, the promoter of this thesis. I am in debt for the time he has spent with me in almost endless discussions. I highly appreciate the fruitful and constructive meetings, during which he provided me with valuable insights. I wish to express my sincere gratitude to him.

My thanks go to the other members of the reading committee, Prof. Dr. Carlos De Backer and Prof. Dr. Karel De Vlaminck, for their comments, which have improved the quality of this text, and to the other members of the jury for taking the time to read the text.

It is a pleasant task of acknowledging the help of colleagues and friends of the Software Development Methodology research group. But very special thanks to Geert Delanote and Pieter Bekaert for the discussions and suggestions on many obscure topics, including football and politics.

Special thanks and appreciation goes to my family and friends for their encouragements. By continually asking me about the date of defence of this thesis, the social pressure was kept high enough not to give up.

Last but by no means least, I want to thank Katelijne, my spouse and anchor point. My life would never have been the same without her.

Contents

Chapter 1

Introduction

Analysis is one of the first phases in software development. During this phase, the functional requirements of the software system to be developed are specified. This specification is built in terms of facts that have been observed in the problem domain or business domain. The problem domain is defined as that part of the real world that is of relevance for the envisaged system.

The ***object-oriented*** paradigm is a software development approach covering all phases of the life cycle of a software system, including analysis. The object-oriented approach focuses on identifying and interrelating objects with their corresponding characteristics and behaviour. In analysis, this result in a model that describes all relevant real-world objects, including their business rules.

A ***pattern*** is a description of a solution to a recurring problem within a certain context. A pattern must describe *how* it resolves the stated problem and *why* it is a good solution. We need both arguments to convince us that the pattern is neither pure theory nor blindly following others [App97].

It is often useful to study how problems should not be solved. Understanding a bad solution can help to determine the good ones. An ***anti-pattern*** is the description of a bad solution to a recurring problem. An anti-pattern must describe *why* it is not a good solution.

In this chapter, the title of this thesis will be further unravelled. In section 1, the purpose and the need for an analysis phase is described and positioned in the software life cycle. In section 2, the object-oriented approach for analysis is briefly introduced and its main characteristics are discussed. Section 3 reveals some major problems and difficulties with current practices of

object-oriented analysis. In section 4, the goals of this thesis are given. In the last section, an overview of the thesis is given.

1 Software Engineering

In software engineering, as for other engineering disciplines, the development of a system is divided into a number of activities. Analysis, ***design***, ***implementation*** and ***deployment*** are common names for some of the phases in the software life cycle. The exact number and naming of the phases may vary between the different life cycle models. In the next subsections, the different phases are positioned in the software life cycle. The emphasis is on the analysis phase.

1.1 Analysis

1.1.1 The Purpose of Analysis

The major purpose of analysis is to establish the functional requirements raised by the clients and to be solved by the system. As promoted eloquently by Jackson [Jack83, p.4], a developer should begin by creating a model of reality, followed by a description of the functional requirements that is based on the created model of reality. Jackson argues that this principle promotes extendibility, preciseness and communication. We claim that functional requirements cannot be precisely described without detailed knowledge of the real world that is relevant for the envisaged system. If this knowledge is absent, a designer cannot be blamed that important aspects in the ultimate system are overlooked. For instance, in such a case it is possible that the ultimate system violates important business rules. In this thesis, it is assumed that the relevant real-world knowledge and the functional requirements must be completely and formally specified. The results of the analysis activity are expressed in a ***conceptual model***. These results are primarily intended as input for the design phase.

1.1.2 Analysis is about real-world Facts and functional Requirements

The analysis phase concerns the specification of real-world facts and functional requirements. During analysis, the transition into a software system is not (yet) at stake [Stee00, p.2]. In other words, analysis is about real-world facts in the problem domain and not about software or hardware facts in the system domain. More precisely, analysis deals:

- with real-world objects such as persons, accounts, cars and books and does not deal with possible software representations of real-world objects such as software classes and software objects nor with database tables and records;

- with structural real-world facts such as the balance of an account and the name of a person and not with possible representations of such properties in terms of attributes of classes or fields of database tables;

- with behavioural real-world facts such as the withdrawal of an account and not with queries updating databases or with software methods manipulating software objects;

- with real-world business rules such as a rule that the balance of an account must exceed its limit at all times and not with decisions about how business rules can be implemented and enforced in a computer system.

Most real-world facts can exist without the presence of computer systems. For instance, accounts, the balance of an account, a withdrawal of an account and the balance of an account that must exceed its limit are observed real-world facts that already existed before the introduction of computer systems.

Notice that the models that will be presented in chapter 2 to 6 are conceptual models. Consequently, only statements in these models are made about observed facts in the real world and no statements are made about possible hardware or software decisions for the envisaged computer system.

Since analysis also deals with functional requirements, another common name for this phase of the software life cycle is ***requirements engineering***

[Roll92, p.3]. However, there is more to specify than functional requirements. Non-functional requirements are another important aspect of specification.

1.1.3 Analysis cannot be omitted during Software Engineering

Because every computer system is concerned with a certain reality outside itself, analysis of real-world facts is a necessary step in the development of a computer system. How can a solution be worked out without knowing the real world in which the system must operate? Notice that a certain implicit model of the real world is always present in the minds of the developers. The lack of an explicit specification of reality often leads to ambiguities disturbing later stages in system development. Don't suppose too fast that all the real-world facts are known in all detail.

As the demand for complex software systems grows, analysis is gaining importance. As stated in [Fran03, p.XX], a steadily increasing percentage of developers revolve around issues related to modelling the business problems to be solved, rather than the details of writing code. This is not because developers necessarily want to move in this direction. The increasing complexity of business problems that must be solved demands such a requirements-driven approach.

Analysis helps the developers to understand and document the business domain. That "developers do not understand the business" is an often and loudly heard complaint by the client of the software system. Moreover, the later mistakes during analysis are detected, the more costly it is to remedy these problems [Gree94, p.135]. The fundamental role of conceptual modelling in software engineering is nowadays generally accepted by the research community. Accordingly, investigation in conceptual modelling remains essential. This thesis mainly deals with research for adequate modelling constructs and guidelines for analysis.

1.1.4 Adequate Analysis Guidelines and modelling Constructs are needed

In order to model precisely and completely all envisaged functional requirements and the related real-world facts, a developer must be offered an adequate conceptual modelling language and an adequate conceptual modelling method. It is a blind assumption, that a language that is appropriate for design purposes is also appropriate for analysis purposes. In chapter 2 to 6, analysis constructs and guidelines are offered for representing all kinds of real-world facts, such as objects, structural and behavioural facts and business rules.

1.2 Design

The major purpose of design is to decide how the requirements resulting from analysis and the non-functional requirements can be implemented in terms of software. The decisions will be based on software quality factors and non-functional requirements [Beka02, p.35]. Consequently, the design phase concerns the description of a software product. The result of the design activity is expressed in a ***computational model*** [Dies00, p.4], also referred to as a design model. Although this thesis mainly deals with analysis, the relationship between the analysis phase and the design phase will be discussed to some extent.

1.3 Implementation and Deployment

The implementation phase deals with coding the software system. This phase carries out the findings of analysis and design and results in executable code. During deployment, the software system is brought into use. These two last phases of the software development cycle fall outside the scope of the thesis, but the applicability of some well-known programming techniques at the level of analysis will be evaluated.

2 An Object-Oriented Approach for Analysis

2.1 Principles and Concepts

The primary modelling construct of the object-oriented approach, namely objects, is said to match well with the real-world environment [Hubb98, p.22]. As observed in [Shlae88, p.9], the world is full of objects or things. The complexity of a real-world problem is conquered by subdividing it into smaller parts. In the object-oriented approach these smaller parts are objects. Booch calls this process ***object-oriented decomposition*** [Booc94, p.17]. Prior to the object-oriented approach, structured analysis (and design) subdivided the problem into functions. This process was called functional decomposition. The data-oriented approach emphasizes data structures.

Contrary to the function-oriented approach and the data-oriented approach, the object-oriented approach places equal emphasis on modelling ***static and dynamic properties*** in order to achieve an integrated result [Roll92, p.12]. Static properties are structural real-world facts while dynamic properties are behavioural real-world facts. Whereas static properties deal with the state of an object at any given moment in time, dynamic properties deal with changes of this state over time.

The object-oriented paradigm introduces powerful modelling constructs such as ***generalization-specialization***, ***polymorphism*** and ***encapsulation*** [Stee02, p.18]. Generalization-specialization allows the analyst to specify an “is-a” relationship between objects. At the level of analysis, we define polymorphism as the capability to reason about a superset of objects, and to further refine such reasoning for subsets of this superset. Both constructs are further discussed in chapter 6. The basic idea behind encapsulation or information hiding is to conceal irrelevant details from the potential users of a given element. This construct results in some basic patterns that are introduced in chapter 2.

2.2 Current Practices

The object-oriented representation of real-world facts is generally accepted as an appropriate foundation for conceptual modelling [Green94, p.136]

[Coad90, p.34] [Stee00, p.1] [Past01, p.508]. This does not prevent that considerable difficulties with object-oriented analysis are reported [Hoyd93] [Embl95] [Sim00] [Opda02]. One of these problems is that most modelling languages and analysis methods are strongly influenced by design and implementation issues. In section 3, the most important problems in this area are described, and some solutions are suggested.

These problems perhaps explain why in practice the specification of the problem domain is often completely or partially omitted, although researchers recognize its crucial role in software development. As stated in [Said03, p.5], by skipping analysis the developer fails to bring the requirements as a central issue into design. These problems perhaps also explain why the analysis phase is poorly understood and a large number of errors can be found in current industrial practice and publications [Fern00, p.183].

We are convinced that the fundamentals of the object-oriented approach, such as for instance object-oriented decomposition, offer significant advantages over other approaches. However, we do recognize that problems still exist in applying an object-oriented approach during analysis (as well as during design and implementation). In this thesis, the traditional object-oriented approach is adopted as far as it permits the analyst to adequately obtain the major goals of analysis.

In accordance with the adoption of the object-oriented approach, the ***Unified Modeling Language*** (UML) [OMG01] will be used and evaluated as a language for conceptual modelling. Nowadays, the UML is the de facto standard for object oriented modelling both during analysis and design. As of this writing, the current version of the UML is 1.4. When this thesis refers to the UML, it refers to version 1.4 unless otherwise stated.

3 Problems with Object-Oriented Analysis

In this section, some of the problems with object-oriented analysis in software development are discussed in more detail. In subsection 3.1, problems related to the fuzzy line between analysis and design are discussed. Subsection 3.2 then outlines the lack of adequate constructs for conceptual

modelling. Throughout this text, we will work out suggestions for better concepts. Subsection 3.3 discusses the need for more guidelines in conceptual modelling. Even experienced analysts need some guidance in how to model real-world facts in the most appropriate way. In subsection 3.4, we report on the lack of proper quality criteria for conceptual modelling. Finally, subsection 3.5 presents some of the problems dealing with the transition from analysis to design.

All of these problems are heavily interrelated. As an example, the fact that most modelling constructs offered by the UML are more suited to build computational models than conceptual models, amplifies the vague border between analysis and design. As another example, the lack of practicable quality criteria strengthens the lack of conceptual modelling guidelines.

3.1 The Boundary between Analysis and Design

In traditional object-oriented software development, the separation between analysis and design is vague. In most cases, the exact boundary is hard to determine. Analysis and design activities are considered to exist on a continuum. Development gently flows from more analysis-oriented activities to more design-oriented activities. The vague border is often viewed as an advantage that permits a smooth transition from analysis to design. Some people even give explicit warnings that a debate about the terms “analysis” and “design” can lead to terminology wars. They claim that trying to be rigid about what constitutes an analysis step versus a design step is not very useful and may even be contra-productive [Larm98, p.15].

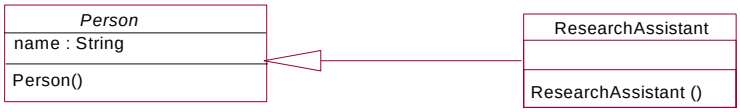
We believe that the vague border between analysis and design must be avoided because of two fundamental disadvantages:

- Due to the vague border between analysis and design, the first and fundamental principle of description, which is to identify clearly the subject of what is being described, is violated [Jack02, p.5]. If a developer does not know if the software development process is at the analysis phase or design phase, it is impossible to answer what the model elements represent. In particular, it will be unclear whether software artefacts are being modelled, or real-world facts. If there is no clear line

between analysis and design, design models can be read as analysis models and vice versa, which obviously can lead to misinterpretations. In such cases, there is the often unspoken assumption that all descriptions are about the software domain [Jack02, p.6] and the problem domain is neglected.

- Lacking a proper line separating analysis from design, the principle of separation of concerns is also violated. An analyst's main goal is to capture real-world knowledge and does not want to capture any design issues. Separation of concerns is a principle that is at the core of software engineering. It refers to the ability to identify, and manipulate those parts that are relevant to a particular concern or purpose [IBM03]. As stated in [Hoyd93, p.243], the burden of the translation from the problem domain to the software domain should not be put on the analysis phase itself, but on transition from analysis to design.

We believe that a strict border must be defined between analysis and design during software development. During analysis, only real-world facts will be represented. Consider as an example the conceptual model in Figure 1. A model element such as the class `Person` represents the set of all persons living on mother earth. At the level of analysis, the class does not represent software objects registering some information about some persons. As a consequence, the constructor `Person()` represents the birth of a person and not the registration of some personal data in an information system.



context Person :: Person(name: String)
post: self.name = name

context ResearchAssistant :: ResearchAssistant(name:String)
post: Person(name)

Figure 1 The property of real world modelling

The property of real-world modelling forces the analyst to focus on the problem domain instead of on the software domain. Although real-world modelling is advocated by many authors [Macg92, p.9] [Brow02, p.100], real-world modelling and its consequences are not fully applied. As observed in [Jack95, p.3], object-oriented analysis is often accompanied by fine words about modelling the real world but when the descriptions are examined in more detail it can be concluded that they are descriptions about programming objects, pure and simple. As an example consider the class `ResearchAssistant`. At the level of analysis, the specification of the constructor `ResearchAssistant()` in Figure 1 is invalid. Some persons are born as research-assistants. This does not reflect and is in contradiction with the real-world knowledge. At level of design, the specification can be valid. A new software object research-assistant is created and consequently also a software object person.

3.2 The Modelling Constructs

As a standard modelling language, the UML is often used for object-oriented analysis, but the constructs offered by the UML (and most other object-oriented modelling languages) to perform analysis are mainly derived from object-oriented design and object-oriented programming. The derivation of modelling constructs for analysis purposes from design languages and programming languages causes several problems:

- A substantial problem in object-oriented analysis is that most modelling languages, such as the UML, are more suitable to build computational models than to build conceptual models. As stated in [Gree94, p.144], it is not wise to design requirements modelling languages by merely adopting programming language ideas. Due to the different goals of analysis and design, it is a very blind assumption that modelling constructs, which are convenient for design and programming, are by definition also adequate for analysis. As an example, are invariants for classes and preconditions for methods adequate constructs to represent business rules at the level of analysis? This topic is discussed in chapter 5. As another example, are attributes adequate constructs to model

structural aspects at the level of analysis? This topic is further discussed in chapter 2.

- In the UML, a definition for a construct is given that should be adequate for analysis as well as for design purposes because the UML claims to cover both phases of the software development cycle. In the UML, this leads to definitions that are either very abstract or to definitions that are rather design-oriented. The problem of a multi-purpose definition for UML constructs is also observed in [Opda02, p.64]: "each definition in the UML glossary is a compromise between different goals, very often promoting technical issues at the expense of representing the problem domain".

- Because the UML constructs are derived from design and programming constructs, three of the four general deficiencies with conceptual modelling constructs, identified in [Wand93], are likely to occur. These deficiencies are: ***construct redundancy***, ***construct excess***, ***construct deficit.*** Construct redundancy occurs when more than one construct can represent the same type of real-world facts. Construct excess occurs when a construct does not represent a type of real-world facts. Construct deficit occurs when a type of real-world fact cannot be represented by any of the modelling constructs in the language. The fourth deficiency is ***construct overload*** and is less likely to occur. Construct overload occurs when the same construct represents several types of real-world facts.

- Perhaps the most difficult aspect of analysis is avoiding software design [Davi93]. Many so-called analysis methods are really design methods because their constructs are more appropriate for design than for analysis. This corrupts the analysis phase by introducing preliminary design decisions. Many conceptual models expressed in the UML are oriented towards system implementation. Lots of design decisions are taken during the development of a conceptual model. As observed in [Embl95, p.19] and in [Hoyd93, p.244], object-oriented analysis as preached and practiced is more preliminary design than a representation of real-world facts, and is rather target-oriented or implementation-oriented than problem-oriented. As stated in [Jacks95, p.1], the inability to describe the problem domain explicitly has been one of the most glaring deficiencies

of software practice and theory. Many development methods claim to offer an analysis of the problem at hand, when in fact they only offer an outline of a solution, leaving the problem itself unexplored and unexplained.

3.3 The Modelling Guidelines

The UML is a modelling language not a modelling method. It does not guide the software engineer in how to perform object-oriented analysis and design. In the current state of the art, there exist no generally accepted guidelines how to use UML for modelling a real-world system [Ever01, p.355]. Consequently, there is a need for a method to guide the analyst how to specify requirements and real-world facts in the most adequate way.

Due to the lack of guidelines, there exist several alternatives to model the same set of real-world facts. Choices usually imply that there are some alternatives that are superior to others, and that several alternatives may exist that model the same real-world facts in an equivalent way. We believe that it is not the task of the analyst himself to evaluate different alternatives; this is the main task of a proper method for object-oriented analysis. An analyst must be able to focus on the real-world facts and not on modelling alternatives. A method should guide the analyst to the better alternatives. Furthermore, such guidelines will lead to standardization in conceptual modelling.

One could argue that patterns guide the developer by describing the best solutions for often recurring problems. However, in the current state of the art, most patterns in object-oriented software development, and definitely the best-known ones, are situated at the design stage [Gamm95]. Examples of analysis patterns can be found in [Fowle97, pp.15-237] [Erik00, pp.187-352] [Dsou99, pp.575-606] [PLoP] [Fern00] [Joha98]. Existing patterns at the level of analysis have several shortcomings:

- Most analysis patterns describe how to model structural aspects and neglect behavioural aspects and/or business rules. Most patterns do not describe *how* to model behaviour or restrictions. At best, they merely suggest adding behaviour and restrictions to structural aspects, without

proper guidelines concerning how to express them. The reason for this is that conceptual models are often restricted to modelling structural aspects only. In that sense, conceptual modelling comes close to information modelling, an established approach in the area of databases.

- Most analysis patterns are domain-dependent. A method for object-oriented analysis must offer guidelines that can be used in all domains. We do agree that most analysis patterns can be abstracted to some degree and become less domain-dependent, but they still are not applicable in all possible domains. As an example [Fern00], an analysis pattern for a hospital treating patients can be abstracted to organizations treating things. Obviously, the more general pattern can be re-used in conceptual models for other domains, such as repairing cars or maintaining computer systems. The abstracted pattern is however still domain-dependent. Moreover, specific knowledge about particular domains, such as detailed contracts for defect computer systems, must still be worked out explicitly. These more abstracted patterns could be considered as frameworks.

- Another technique, aside abstraction, of many analysis patterns is to introduce meta-level constructs in a conceptual model. For instance, the analyst could introduce the class `Attributes` in a conceptual model. These patterns claim to promote flexibility and extendibility. On the one hand, we do not always reject this technique, on the other hand, the ultimate flexibility and extendibility cannot be found in proposing the meta-model as conceptual model for a particular problem domain.

- Most analysis patterns take the modelling language as given and do not evaluate the used modelling constructs. In fact, some patterns are introduced for expressing real-world facts in terms of the limited power of the modelling language. With more proper supporting concepts, there would have been no need for such patterns. The possible problems with modelling constructs were already described in the previous subsection.

- Some analysis patterns are in fact design patterns. Consider as an example the “temporal patterns” proposed in [Fowl]. These patterns describe how to model the history of changes. The “audit log” pattern

proposes to record the history of changes in a log file. This is clearly a design decision.

3.4 The Modelling Quality Criteria

It is very difficult to build guidelines for conceptual modelling if it is impossible to define quality criteria for conceptual modelling. As stated in [Sisa02, p.414], the literature devoted to conceptual modelling quality criteria is limited, in particular for UML schemas. Examples of quality criteria for conceptual modelling can be found in [Lind94] [Asse96] [Sisa02]. Existing quality criteria reveal several shortcomings. These shortcomings are interrelated and presented in next paragraphs:

- Many quality criteria are vaguely defined. Consider as an example the quality criterion *feasible completeness* [Lind94]. Feasible completeness is defined as a state in which no real-world fact can be found that is not modelled, and where the additional benefit to introduce this fact in the model exceeds the additional drawbacks of including it. We agree with the authors that total completeness is not reachable, but the problem remains that there exists only a vague comprehension of the terms “additional drawback” and “additional benefit”.

- Many quality criteria are not measured appropriately or are difficult to measure. Examples of ***conceptual modelling metrics*** can be found in [Poel98] [Gene02] and in [Sisa02]. Consider as an example the quality criterion *simplicity*. In [Sisa02, p.419], simplicity is measured in function of the number of relationships between classes in a conceptual model. Consequently, a conceptual model with one relationship and two classes is more complex than a conceptual model with thousand classes and no relationships. We believe that this is an example of a badly measured quality criterion. Furthermore, a model representing a lot of real-world facts is more complex than a model representing only a few real-world facts. But measuring this complexity as such does not help an analyst in building quality models. An analyst must model a given set of real-world facts and cannot decide to model only a certain subset of the given set. If the analyst cannot replace a model element by other model elements

representing the same real-world facts, it is invalid to simply remove this model element in order to increase simplicity.

- Some quality criteria are questionable. Consider as examples the quality criteria of *homogeneity* and *explicitness*. A class represents a set of objects and should be homogeneous. According to [Asse96], this means that classes may not contain highly dissimilar objects. From this perspective, the quality criteria homogeneity is not inline with the construct of generalization. The important construct of generalization in the object-orientation paradigm leads to classes containing more dissimilar objects. On their turn, these classes could be generalized, which leads to classes with even more dissimilar objects. On the other hand, specifications of a class, which apply on all objects, are preferred against specifications of class, which only apply on a particular subset of object of that class. This preference can be also interpreted as "classes must be homogeneous". From this perspective, we agree that classes should be homogeneous. Explicitness means that information about the problem domain should be represented at the class level and not at the object level. One could question why it is wrong to introduce an object with specific properties in a conceptual model? Why can an analyst specialize classes into subclasses, which represent two objects, but why can he not specialize a class into one object? This kind of specialization is further discussed in chapter 6. Of course, a large number of objects will not be modelled directly via objects; classes remain a dominant construct in conceptual modelling.

- Some quality criteria are obvious. Of course, this is not really a shortcoming, but these obvious criteria are insufficient to guide an analyst to "good" models. Consider as an example the criteria *syntactic correctness* and *semantic correctness*. A model is syntactic correct if all specifications adhere to the modelling language rules. A model is defined as semantic correct if all specifications describe the observed real-world facts in a correct way.

- Some quality criteria deal with understandability of the model by its possible users, such as domain experts, end-users, clients and designers. Once again these quality criteria are not a shortcoming, on the contrary,

but from our point of view a conceptual model is an artefact made by the analyst and is not *necessarily* a communication tool. In this thesis, no statements are made about the way an analyst should communicate with his audience and about the appropriateness of conceptual models as communication tool. Obviously, if the audience of an analyst doesn't understand conceptual models, the analyst is forced to search for other ways of communications, but an argument as "this conceptual model is too complex to be understood by a possible user" is not a valid argument against that model as such. This only means that the analyst must search for other ways of communication than the conceptual model at hand.

3.5 The Transition into Design

The transition into design is sometimes viewed as a process refining the model step by step. During design, more technical details are added to the model. We have already stated that it is a blind assumption that design constructs are also suited for purposes of analysis. The opposite is also true: elements populating conceptual models are not necessarily appropriate building blocks for computational models. For instance, a certain class or a certain attribute that are present in a conceptual model, are not necessarily present in a computational model. Moreover, typical constructs for analysis, such as class constraints (presented in chapter 5), cannot remain in a design model per definition.

The transition process from conceptual models into design models and code is complex and not fully understood. Consequently, the connections between analysis artefacts on the one hand, and design artefacts and code, on the other hand, are too loose. Traditionally, the transition is done by hand and not automated. Conceptual models are often looked at as documentation only and as a productivity loss. As observed in [Klep03, p.5], this is one of main reasons why documentation, and thus conceptual models, is often not of very good quality. Some people argue that in the object-oriented paradigm the transition from analysis to design is an easy process. This is often because their conceptual models don't describe real-world facts, but are rather high-level design models.

We are in favour of automated transformations of analysis models into design models and/or programming code. Automated transformations are a key element in the MDA framework [OMG03]. The MDA approach tries to raise the level of abstraction for producing programming code. From this perspective, the analysis level is the ultimate level of abstraction. Automated transformations from conceptual models into design models fall outside the scope this thesis.

4 Goals of the Thesis

The goal of this thesis is threefold. The objectives are briefly introduced in next three paragraphs and further worked out in the next three subsections.

- A first objective is to evaluate the adequateness of the UML constructs to perform analysis. This objective is limited to the UML constructs for building class diagrams, object diagrams and uses case diagrams. These diagrams can be typically used for analysis purposes. Developers also most often use these diagrams when the UML is adopted [Zeic02]. If no adequate constructs or semantics for these constructs are available in the UML, better constructs and semantics will be proposed.

- A second objective is to offer the analyst patterns and anti-patterns for conceptual modelling. These patterns guide the analyst in building quality conceptual models and allow him/her to focus on the problem domain.

- The third objective is to define principles for conceptual modelling. These principles will help to evaluate the proposed modelling constructs and guidelines and can be considered as quality criteria for conceptual modelling.

4.1 The UML and Conceptual Modelling Constructs

An analyst needs an adequate modelling language, offering a set of adequate conceptual modelling constructs. Each of these constructs must allow the description of a certain type of real-world facts. In other words, the use of

these constructs represents a real-world semantics. In this set of constructs, no construct redundancy, construct excess, construct deficit and construct overload should occur. We believe that this modelling language should be as much as possible in line with the UML, the de facto standard modelling language for object-orientation.

The UML provides a built-in extension mechanism, which allows defining ***profiles*** [OMG01, p.2-74]. A profile is defined for specific purposes and is a coherent set of compliant UML-extensions, defined by stereotypes, constraints and tags. A major problem is that possible extensions may not be in conflict with the predefined UML constructs; it is only possible to refine predefined UML-constructs. From the UML's point of view, this is logic: if it would be possible to contradict existing UML constructs, then there would be no standardization anymore. Another problem is that the extension mechanism of the UML is rather primitive.

In the next paragraphs, three directives are proposed in order to ameliorate the UML for conceptual modelling. In this thesis, these directives will be handled in a partial way. These directives are:

- The UML constructs that are adequate for analysis purposes must be clearly identified. This is only a subset of the UML constructs. Other UML constructs are defined to be inadequate for analysis purposes.

- The use of each of these useful UML constructs must represent a well-defined real-world meaning. Different semantics for constructs that are used during analysis and design is needed because of the different goals of both phases in the software lifecycle. Probably it would be better to use different names for constructs with different semantics.

- Adequate constructs for analysis purposes that are not offered by the UML must be clearly identified. If possible, these constructs could be added to the UML by making use of a profile. Otherwise, these constructs could be adopted in a later version of the UML.

4.2 Patterns and Anti-Patterns

Aside adequate modelling constructs, an analyst needs adequate modelling guidelines. In the next chapters, patterns and anti-patterns for analysis purposes will be proposed. These patterns and anti-patterns are situated at the meta-model level. As stated in [Fowle97, p.298], ***meta-model patterns*** describe modelling languages rather than the models themselves. Meta-model patterns describe the use of constructs in modelling languages, whereas "normal" patterns or ***model patterns*** describe useful models [Devo02, p.478].

In the UML, a four-layer meta-model architecture is defined. These layers are: the user objects layer (M0), the model layer (M1), the meta-model layer (M2) and the meta-metamodel layer (M3). The primary responsibility of the meta-model layer is to define a language for specifying models, whereas the primary responsibility of the model layer is to define a language that describes a problem domain [OMG01, p.2-4]. Meta-model patterns operate at the meta-model layer, whereas model patterns operate at the model layer.

Meta-model patterns can give rise to the introduction of new constructs and the rejection of existing constructs, e.g., a new construct for modelling the history of values is introduced and the use of an association class is rejected in [Fowle97, p.301-305]. Meta-model patterns investigate which constructs are needed to specify real-world facts, and how they can be used. Meta-model patterns describe the use and adequateness of meta-level constructs such as attributes, queries, associations, preconditions, invariants, classes and generalization relationships. An advantage of meta-model patterns is that these patterns are not domain-dependent.

The patterns worked out in this text are not only meta-model patterns but also ***analysis patterns***. Analysis patterns describe solutions to recurring problems in the context of conceptual modelling. We want to underline that these patterns are only valid during object-oriented analysis. During object-oriented design, other, design-oriented patterns are needed [Gamm95]. Meta-model patterns for analysis help to model reality and are part of an analysis method. A good analysis method imposes rules on the constructs it offers. An analyst must focus on the problem domain rather than to concentrate on possible modelling constructs or alternatives to model reality. A decision of

an analyst on *how* to represent an observed fact indicates a lack in the analysis method. We believe that the ultimate goal of an analysis method is to guide the analyst in specifying precisely all relevant real-world facts without any design or modelling concern.

The patterns proposed in the next chapters will deal with structural aspects as well as with behavioural aspects, with business rules and with functional requirements. The proposed patterns will be described using the following template: a problem description, possible anti-patterns or bad solutions followed by a solution that defines the pattern. Each pattern will be illustrated with examples.

4.3 Principles for Conceptual Modelling

In this subsection, some principles for conceptual modelling will be proposed. The proposed analysis patterns in the next chapters will be evaluated with the help of these principles, however not all principles can be always respected.

4.3.1 The Principle of Completeness

Given a finite set of real-world facts S, the ***principle of completeness*** states that *all* real-world facts of S must be represented. In [ISO82] and [Borg85], the need to model behavioural aspects and restrictions was already emphasized. Other authors propose to postpone such kind of "details" until the design or even until the implementation phase [Peet01, p.88]. In contrast with these authors, we do not see why it would be unimportant to model a certain real-world fact f1, which is an element of S, if it is important to model another real-world fact f2, which is also an element of S. Problems will arise as soon as a design model is derived from a conceptual model in which certain facts have been omitted. For instance, the designer cannot be blamed that the ultimate system violates a given business rule, if the latter has not been included in the conceptual model. The designer has to focus on design decisions and not on identifying business rules.

This principle must be extended towards the functional requirements. A conceptual model must describe all the functional requirements imposed on the target system. Notice that this can imply that the set of real-world facts S must be extended, because the specification of functional requirements is based on the specification of the real-world facts. For instance, it is impossible to specify a functional requirement that returns the balance of an account given the account number when there is no notion of balance, account number or account.

We further assume that the set of real-world facts and the set of functional requirements that must be represented are given and correct. This thesis does not cover the knowledge acquisition process. During this process, the analyst should discover and capture knowledge. Such knowledge may be found in existing computer systems, in documentation and in interviews with domain experts. This thesis only deals with the conceptualisation process, which concerns the formalization and representation of discovered knowledge.

Notice that we clearly distinguish between real-world facts and functional requirements. Real-world facts are statements about observations of the reality. Functional requirements are statements about what the target system should offer to its end-users. For instance, a statement such as “all accounts have a balance and an account number” is a real-world fact. A statement that “a system must be able to return the balance given the account number” is an example of a functional requirement.

4.3.2 The Principle of Abstraction

A conceptual model representing some part of the external world may become rather complicated due to the inherent complexity of the real world. On the other hand, an application is only interested in a fraction of all details of the external world. Generally, abstraction will help the analyst to bridge the gap between the overwhelming complexity of the real world and the aim for simplicity in software engineering.

The ***principle of abstraction*** states that *only* the real-world facts of the given set S may be modelled. In other words, the analyst cannot be forced to

further investigate the problem domain in order to be able to model the given set of real-world facts. For instance, an analyst is not obliged to specify the upper bound multiplicity of an association end, if that upper bound is not relevant. Assume as an example that a supplier must have at least three bank accounts. The principle of abstraction demands that this fact can be modelled without the need for the analyst to investigate the maximum number of bank accounts a supplier can have.

Some analysis methods are in favor of more elaborated models. Consider as an example the object-oriented analysis method EROOS [Steeg00, p.79]: "The abstract model is a compact way to talk about the given characteristic. Although very attractive at first sight, a compact model greatly compromises reusability and extendibility. Indeed by focusing on the more compact model, the software engineer delimits its interest to the system he has to build at this very moment. In this way, the software engineer is not anticipating future extensions to the system, nor is he trying to build a model that can be reused in other applications operating within the same external world." These methods impose modeling constructs and modeling rules, e.g. no n-ary associations, no integer or Boolean attributes, which lead to more complex models. We believe that the fact that a developer can delimit its interest to the given set of real-world facts is an advantage: the developer can focus on the given real-world facts. Furthermore, this principle leads to simpler models. A software engineer cannot anticipate all possible future extensions. It is the task of the object-oriented analysis method to foresee an easy transition into a more extended model. This easy transition is guaranteed by the principle of extendibility, which is described in section 4.3.3.

Obviously, the principle of abstraction also applies to the functional requirements: the analyst should only model the given set of functional requirements. During the specification of the functional requirements the principle of abstraction can be further strengthened: it states that *only* the *relevant* real-world facts of the given set of real-world facts S may be modelled. Whether or not a real-world fact is relevant depends on the functional requirements of the system to be developed. If a specific real-world fact becomes involved in one of the services to be provided by the system, it must be part of the conceptual model for that application. Consider as an example a functional requirement that returns the balance of an

account given the account number. The real-world fact that an account has a balance and an account number is relevant. The real-world fact that an account has a limit is not relevant as far as this functional requirement is concerned.

Notice that the assumption that a well defined set of real-world facts is given during conceptual modelling does not always hold in a real-world situation. In such a case, an analyst can only apply the principles of completeness and abstraction at the moment the functional requirements are given.

4.3.3 The Principle of Extendibility

The ***principle of extendibility*** states that it must be possible to extend a model with a set of real-world facts S′ without modifying existing specifications. This principle promotes extendibility and reusability. It also applies to functional requirements.

Consider the example outlined in Figure 2. Assume that every book has a certain circulation, expressing its number of printed copies. This real-world fact is represented in the initial model. Assume that after a while the model must be extended with information about the printed copies of each book. This is modelled in the extended model, by introducing the additional class Copy. The principle of extendibility is not violated in this case: the extension is modelled without changing existing specifications.

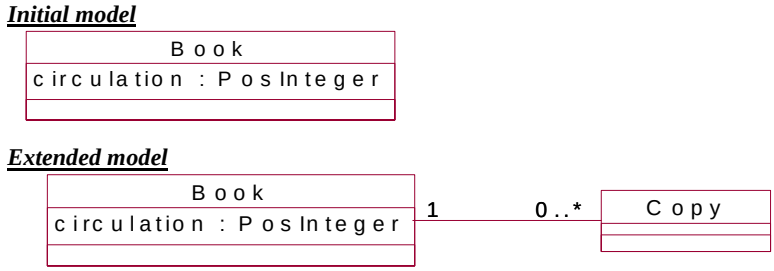


Figure 2 The principle of extendibility

4.3.4 The Principle of Preciseness

The ***principle of preciseness*** states that all real-world facts in the given set S must be formally modelled; natural language is only allowed in the comments. Again, this principle also applies for representing functional requirements.

Practice has shown that the use of a natural language always results in ambiguities. The requirement of formal models is in line with the ***Model Driven Architecture*** (MDA) approach, which imposes formal models that can be machine-processed. The MDA is about using modelling languages as programming languages rather than merely as design languages [Fran03, p. XV].

The UML offers the ***Object Constraint Language (OCL)***. The OCL is a formal language using first-order logic and set theory. Despite its formal nature, specifications in the OCL are easy to read and write [OCL01, p. 6-1]. The main constructs offered by the OCL are invariants, preconditions and postconditions. These constructs are thoroughly discussed in chapter 5.

4.3.5 The Principle of No-Choice

The ***principle of no-choice*** states that a given set of real-world facts S can be only represented by just one model. A method should not allow different equivalent alternatives for expressing a given set of real-world facts. This principle also applies on functional requirements. In our opinion, it is not the responsibility of the analyst to decide *how* to model reality. An analyst has to concentrate on the requirements and the real-world facts themselves. A good method has to give answers on how to model a given set of real-world facts in the best possible way. A good method has investigated and evaluated different alternatives. As a result, a method describes guidelines for the analyst how to model reality. Furthermore, this principle promotes standardization. It leads to one conceptual model for one set of real-world facts.

4.3.6 The Principle of No-History

The ***principle of no-history*** states that a conceptual model, which represents the set of real-world facts S and is extended with a new set of real-world facts S', must result in the same conceptual model as when the real-world facts of the union of S and S′ were modelled at once. This principle strengthens the principle of no-choice. A conceptual model must represent real-world facts and shouldn't represent the history of extensions. It also applies to functional requirements.

Consider the example in Figure 3 and Figure 4. Assume that each account has a balance and that each transaction has an amount. A transaction is always linked with one account. The balance of each account equals the sum of the amounts of all transactions on this account. In some methods for object-oriented analysis, the balance of an account can be modelled as a query. *All* these real-world facts are modelled in Figure 3.

context Account :: balance() : Euro
post: result = transaction.amount→sum()

Figure 3 The sets S and S' of real-world facts modelled at once

Assume an initial model that just represents that all accounts have a balance. The initial model can be found in Figure 4. Assume further that after a while this model must be extended with the notion of transactions. The principle of extendibility implies that we cannot replace the attribute balance by a query balance(). This results in the extended model of Figure 4. In this model, the principle of no-history is violated: the extended model of Figure 4 is not the same as the model of Figure 3. Notice that in this example, a problem of construct redundancy occurs. There exist two possible constructs to represent a dependency between the balance of an account and the amounts of all associated transactions: a postcondition and an invariant.

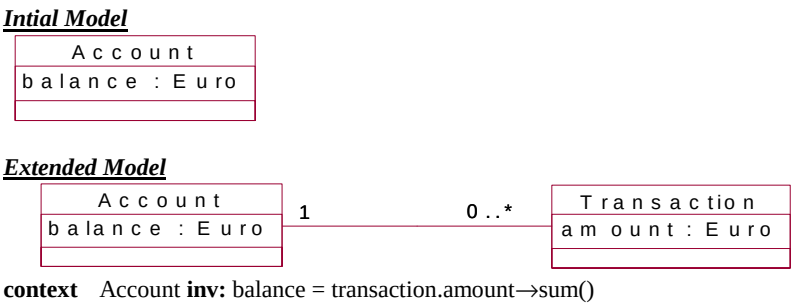


Figure 4 The intial model and the extended model

4.3.7 The Principle of Uniqueness

The ***principle of uniqueness*** states that a real-world fact is modelled only once. This principle promotes adaptability. If the real-world fact changes, the model must only be adapted once. This principle also applies to functional requirements.

5 Overview of the Thesis

This thesis is organized as follows: chapter 2 presents patterns for modelling structural aspects, while patterns for describing behavioural aspects are given in chapter 3. In chapter 4, patterns that are valid for modelling behavioural aspects as well as structural aspects are discussed. Chapter 5 deals with patterns for modelling business rules. In chapter 6, patterns about the use of the generalization are the focus of interest. In chapter 7, a pattern for specifying functional requirements is proposed. Finally, chapter 8 gives some concluding remarks.

Chapter 2

Patterns for Modelling Static Properties

As defined in the previous chapter, static properties are real-world facts that refer to structural aspects of the observed world. These properties deal with the state of an instance at any given moment. Static properties have no side effects, in other words, they do not change the state of the world. Examples of static properties are the balance of an account and the owner of an account.

How can static properties be modelled in an appropriate way? A distinction is drawn between links among objects and links among objects and data values. The distinction between data values and objects and how to model links between them is discussed in section 1 and 2. Section 3 describes how to model links between objects.

The way to model links between data values is not discussed in this thesis. Sections 4, 5 and 6 rather describe the use of data types in conceptual models. Section 4 imposes restrictions on the use of primitive data types and on the use of units as data types. In section 5, user-defined data types are used to restrict the possible set of data values. In section 6, the notion of multiplicity is extended.

In reality, dependencies between static properties can be observed. The relationship between the duration, the start and the end of a meeting is an example of such a dependency. How can such dependencies be specified? Section 7 deals with this question.

It is not sufficient to know how to model static properties and dependencies between these properties. It is also necessary to structure the static properties around the correct model element. How can the suitable model element be detected? These questions are discussed in the sections 8, 9 and 10.

In section 11, the influence on object-oriented analysis of the first and second normal form in relational database design is investigated.

Finally, sections 12 and 13 present two preliminary ideas to promote the adaptability of a conceptual model. These ideas are encapsulation in analysis and modelling dependencies on the model-level. To make these ideas more concrete, they are applied on constructs that represent static properties.

1 Structuring Static Properties

1.1 Problem Description

A static property can be defined as a link between instances. A link is a connection between instances [OMG01, p.2-102]. Consider the example outlined in Figure 1. In this example, accounts have a balance in Euro and an account is said to be positive if its balance is positive. Accounts, Booleans and Euros are said to be instances. The links between accounts and Booleans and between accounts and Euros are examples of static properties.

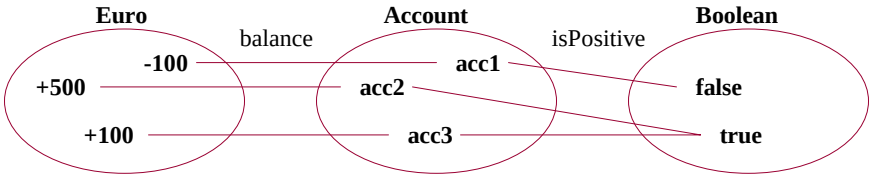


Figure 1 Static Properties

In object-orientation, instances of the same kind will be grouped into a ***classifier*** [OMG01, p.2-28]. A classifier is a model element that represents a set of instances. This process of grouping instances is actually a process of generalization. As instances of the same kind are grouped and represented by a classifier, static properties of the same kind will be also grouped and represented by a certain model element. Preparing generic descriptions that describe many specific items is often known as the ***type-instance dichotomy***.

An important goal in object-oriented conceptual modelling is to represent real-world facts in a structured way. How can static properties be structured in an appropriate way? In object-orientation, model elements are structured around a classifier. Frequently, a distinction is made between two kinds of classifiers: a ***class*** and a ***data type***. During analysis, a class models a set of real-world objects while a data type models a set of data values. In the example of Figure 1, accounts are said to be real-world objects and Booleans and Euros are said to be data values. The static properties outlined in Figure 1 are examples of links between objects and data values.

If a distinction between classes and data types is made, some questions arise:

- How are static properties structured around classifiers? Do we attach a model element, which represent a set of static properties, to data types or to classes or to classes and data types?

- Is the distinction between a data type and a class useful? If the distinction is useful, which criteria exist to make a distinction between objects and data values?

1.2 Anti-Patterns

1.2.1 Structuring Static Properties around Classifiers

A model element, which represents a set of static properties, can be attached to *all* involved classifiers. In such an approach, no distinction between a data type and a class is made. Consider the example outlined in Figure 2. The model element `isPositive` is attached to the classifiers `Account` and `Boolean`. This model element represents the set of links between accounts and Booleans. The model element `balance` is attached to the classifiers `Account` and `Euro`. This model element represents the set of links between accounts and Euros. Notice, that in this example a binary association is used as a model element that represents a set of static properties. The focus of this section is however not on the used model element but on structuring static properties. Other model elements, e.g.

attributes or queries, could have been envisaged. Section 2 and section 3 discuss the appropriateness of model elements to represent static properties.

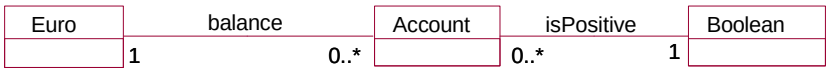


Figure 2 Structuring static properties around classifiers

The method Catalysis [Dsou99, p.81] makes also no distinction between data values and objects. Moreover, attributes and associations can be used to model links between objects. Both model alternatives are equivalent. It is the task of the analyst to evaluate these alternatives.

In our judgement, making no distinction between data types and classes results in more complex and less intuitive conceptual models. From our object-oriented point of view, we consider objects and classes as central constructs to build conceptual models. We want analysts to focus on objects and classes rather than on data values and data types. Data values are considered to be better known because they have always been there. Model elements, which represent a set of links between objects and data values, will not be attached to all involved classifiers. Consequently, no associations will be used to model links between data values and objects.

1.2.2 Structuring Static Properties around Data Types

A model element, which represents a set of static properties, can be attached to all involved data types. In such an approach, a distinction between a data type and a class is made. Consider the example outlined in Figure 3. The model elements `isPositive` and `balance` are attached to the data type `Boolean`. Notice that in this example an attribute is used as a model element that represents a set of links between objects and data values, but that the focus of this section is not on adequate model elements.

Figure 3 Structuring static properties around data types

In our opinion, this approach is not adequate to structure real-world facts. Firstly, the homogeneity of the model elements attached to the data type is questionable. For instance, links between a Boolean and an account and links between a Boolean and a car will be both structured in accordance with the data type `Boolean`. Secondly, in a complex model, a large number of model elements will be attached to a restricted number of data types. As a result, a relatively small number of data types will contain a very long list of attached model elements. This approach will therefore not lead to a well-structured conceptual model of real-world facts, which is one of the major goals of analysis.

1.3 Patterns

1.3.1 Structuring Static Properties around Classes

A model element, which represents a set of static properties, is attached to all involved classes. In such an approach, a distinction between a data type and a class is made. Consider the example outlined in Figure 4. The model elements `isPositive` and `balance` are attached to the class `Account`. Notice that in this example an attribute is used as a model element that represents a set of links between objects and data values, but that the focus of this section is not on adequate model elements.

Figure 4 Structuring static properties around classes

This approach focuses the attention of an analyst on objects and classes and subscribes the object-oriented paradigm, in which objects (and classes) are central constructs. In [Fiad92, p.135], several advantages, such as better insight, are reported when objects are adopted as the basic building blocks for conceptual schemas. Modelling elements, representing a set of static properties, will be attached to the involved class rather than to the involved data types.

The consequences of this approach are described in the next paragraphs:

- Given a large set of real-world facts, this approach results in more intuitive, simpler and better-structured models compared with other approaches, in which no distinction between classifiers is made or in which the model element is attached to a data type.

- This approach violates the principle of abstraction. If an analyst observes links between instances, he is obliged to investigate if these instances are objects or data values. Furthermore, the analyst is obliged to investigate if any objects are involved in the observed real-world facts. For instance, if links are observed between account numbers and euros, the analyst is forced to model these links indirectly by attaching the model elements `account number` and `balance` to the class `Account`.

- Compared with the approach where static properties are structured around classifiers, only one multiplicity is specified. The default multiplicity is `[1..1]` [OMG01, p.3-43]. In the example, an account has just one balance. No statements are made about the minimum and maximum number of accounts that can have a certain balance. This disadvantage is further discussed and resolved in the pattern about the use of data types.

- Three different kinds of static properties can be distinguished: (1) links between objects, (2) links between data values and (3) links between data values and objects.

1.3.2 Objects versus Data Values

What are the criteria to decide whether an instance is an object or a data value? From our point of view, objects are considered to be brought into being and data values are considered to be perpetual [Stee00, p.62]. Consequently, there exists a time period after and before the creation of an object. An account is an example of an object: it is opened at a certain point in time. On the other hand, data values have always been there and will always be there. The quantity three is an example of a data value: it has existed forever, and will continue to exist forever.

In the UML [OMG01, p.2-113], three other differences between objects and data values are put forward: (1) the state of an object can change while the state of a data value cannot change, (2) objects have an implicit identity while data values have no identity and (3) only queries are applicable to data values. These differences are observed in most programming languages [Macl82]. These differences can be defined by a methodology but they cannot be used during analysis as criteria to decide if an instance is an object or a data value:

- Consider some real-world instances x and y and a link between both. Suppose further that after some time, x is linked with y' and no longer with y. Are the states of x, y and y' changed? If the state of x is changed, then the states of y and y' are also changed. But if x is an account and y and y' are account numbers, we have the tendency to say that only the state of x is changed. Having a changing state is not an objective criterion to decide if an instance is an object or a data value.

- Do the instances x, y and y' have an implicit identity or not? What are the criteria to decide if an instance has an implicit identity or not? As stated in [Dsou99, p.258], the identity of anything is a model constructed for our convenience rather than something inherent.

- Furthermore, why is it impossible to apply an event `withdraw()` on an account number at the level of analysis? Or, why can't the data type `Integer` have an attribute `double`, which represents the property double of an integer?

1.3.3 Links have no Direction

Static properties are links between instances. As it can be observed in the example of Figure 1, links have no direction. It is not because a distinction between classes and data types is made at the level of analysis that a direction is suggested. This distinction is made to structure the real-world facts and not to model directions in links. During design and implementation, links are usually unidirectional from objects to data values. In the UML Reference Manual [Rumb99, p.170], the authors state that a data value has no knowledge of an object and a unidirectional from objects to data values is assumed. This is a rather philosophical or a design argument. Why does an account has knowledge about its links with data values and does a Boolean have no knowledge about its links with objects at the level of analysis?

2 Modelling Links between Objects and Data Values

2.1 Problem Description

In section 1, it was decided to attach a model element, which represents a set of static properties, to the involved classes instead of to the involved data types. In this first section, no statement was made about the model element itself. In this section, model constructs that can represent a set of links between objects and data values, such as attributes and queries, are evaluated.

In the real world, ***information redundancy*** occurs when a real-world fact can be computed in terms of other real-world facts. Reconsider the example in Figure 1. A static property “isPositive” can be computed in terms of the static property “balance”. In a model, information redundancy occurs if a model element can be computed in terms of other model elements. Redundant information raises the question if such information is allowed and represented during conceptual modelling.

At the level of analysis, information redundancy is considered to be useful and important in accommodating multiple viewpoints and in reaching a complete and consistent description of the problem domain [Borg85, p.67].

A goal of analysis is to represent all relevant real-world concepts, even if information redundancy exists. Consider another example. Assume a real-world observation were all accounts have just one balance. The balance of an account is clearly a real-world concept; therefore, it must be represented in the conceptual model even if the balance of an account can be computed in terms of the amounts of all transactions on this account. In this section, model constructs that can represent redundant information, such as derived attributes and queries, are evaluated.

2.2 Anti-Patterns

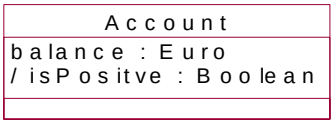
2.2.1 Derived Attributes

In the UML [OMG01, p.3-93], a ***derived model element*** is defined as a model element that can be computed from other model elements. Consequently, a derived attribute is an attribute that can be computed from other model elements. In the UML, an attribute [OMG01, p.2-24] is defined as a named slot within a classifier that describes a range of values that instances of the classifier may hold.

A derived attribute can be used during the analysis phase as well as during the design phase. At the level of design, a derived attribute is introduced for performance reasons. It represents a value that is physically stored in order to avoid a recomputation. At the level of analysis, a derived attribute is included to define a useful name or concept [OMG01, p.2-44].

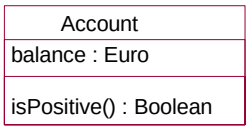
In the UML, information redundancy during analysis can be modelled by a derived attribute or by a query. A query [OMG01, p.3-45] is defined as an operation that does not modify the system state. A query has a return type. Consider the example in the Figure 5. In the first alternative, the set of static properties “isPositive” is modelled as a derived attribute. In the second alternative, these properties are modelled as a query.

Alternative one



context Account **inv:** (balance >= 0 Euro) = isPositive

Alternative two



context Account :: isPositive()
post: result = (balance >= 0 Euro)

Figure 5 Derived attributes versus queries

We reject the construct of derived attributes at the level of analysis. The reasons are described in the next paragraphs:

- The construct of a derived attribute offers no surplus value compared with the construct of a query at the level of analysis. By rejecting the construct of derived attributes, construct redundancy between both model elements is avoided at the level of analysis. Recall that construct redundancy occurs when different modelling elements can represent the same real-world facts. Notice, that this construct redundancy does not exist at the level of design.

- Possible confusion about derived attributes during analysis and design could exist. One could argue that there is clear distinction between the use of derived attributes during analysis and during design. Derived attributes do not represent any decision about data storage at the level of analysis. This is correct, but the problem lies in a possible vague border between analysis and design. When a developer does not know if the software engineering process is at the analysis phase or at the design phase, it is impossible to answer what the derived attribute represents: a piece of stored data or not. By forbidding the construct of derived

attributes during analysis, possible confusion is avoided. In our approach, the use of derived attributes is reserved for the design phase. This helps to make a strict distinction between analysis and design: derived attributes only appear at the level of design.

2.2.2 Queries and Attributes

Queries as well as attributes can be used to model links between objects and data values. The redundancy of one of both constructs during the analysis phase is illustrated in Figure 6. In the class `Account`, one can easily add an attribute `balance` and/or a query `balance()`. In the second alternative, the same set of static properties is modelled more than once. The principle of uniqueness is violated. For this reason, alternative two is rejected. In contrast with the analysis phase, both constructs are not necessarily redundant during the design phase. Temporally, alternative one and alternative three are valid alternatives to model a set of links between objects and data values. In order to satisfy the principle of no choice, other arguments must be considered to make a more rational choice between possible alternatives.

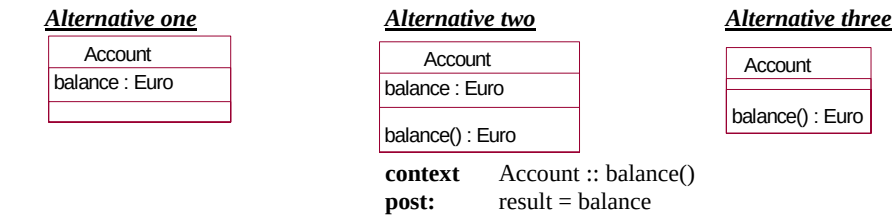


Figure 6 Queries and attributes

In many cases, analysts use queries to model information redundancy and attributes to model non-redundant information. This could be a possible criterion to make a distinction between the use of both constructs and consequently to demonstrate the need for both constructs. On the other hand, one can question the need of making a distinction between redundant information and non-redundant information during the analysis phase. Furthermore, during the extension of a model, non-redundant information

can become redundant. Consequently, an attribute must be deleted and a new query must be introduced. The specification of the class is changed and the principle of extendibility is violated. As an example, consider Figure 7. In the first phase, only a class `Account` existed with an attribute `balance`. In a later phase, a class `Transaction` is added to the model. The balance of an account is the sum of all the amounts of the transactions executed on this account. The attribute `balance` is deleted and a new query `balance()` is added.

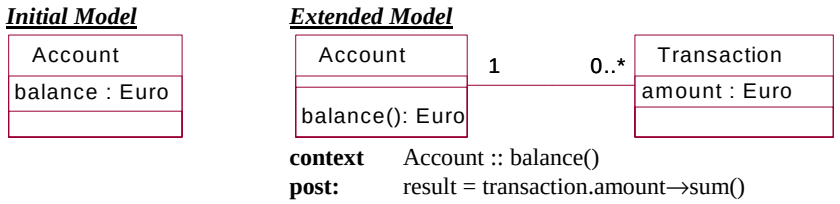
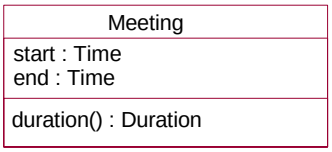


Figure 7 The principle of extendibility is violated

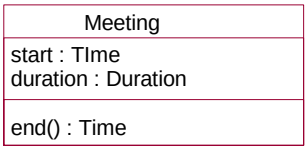
This approach, which uses queries for modelling redundant information, can also lead to different alternatives in case more than one derived model element exists in a information redundancy relationship. Consequently, this approach violates the principle of no choice. Consider the example in Figure 8. A meeting has a start time, an end time and duration. In alternative one, the start time and the end time are modelled as attributes; the duration is modelled as a query. Each combination of two attributes and a query is a possible model. Moreover, in case of a vague border between analysis and design, each of the three alternatives suggests a data representation. For instance, the second alternative suggests to design a class `Meeting` with the attributes `start` and `duration` while, in alternative three, the analyst suggests to design a class `Meeting` with the attributes `end` and `duration`. Decisions about data representation must be postponed until the design phase.

Alternative one



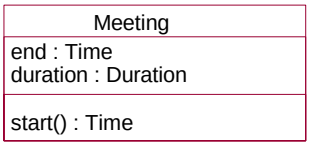
context Meeting :: duration()
post: result = end – start

Alternative two



context Meeting :: end()
post: result = start + duration

Alternative three



context Meeting :: start()
post: result = end – duration

Figure 8 The principle of no choice is violated

Borgida [Borg85, p.67] already promoted to allow the user to treat redundant and non-redundant information in a syntactically uniform manner at the conceptual level. When no different constructs are used to model redundant and non-redundant information, extendibility is better supported: the analyst must only add a specification to represent the redundancy relationship between the model elements and must not change existing specification.

At the level of analysis, the construct of attributes and the construct of queries are not allowed simultaneously. The reasons are:

- The presence of construct redundancy.
- The principle of extendibility is violated in case a query represents redundant information and an attribute represents non-redundant information.
- The principle of no choice is violated in case more than one derived model element exist in an information redundancy relationship.

2.2.3 Only Attributes

In the UML, an attribute can only be attached to *one* class, it cannot have parameters and it can have only *one* classifier as type. Therefore, it can be only used to model a set of binary links and it cannot be used to model a set of n-ary links. Consider the example outlined Figure 9. A product has a price at a certain date. The price of a product can be the same at different dates. This real-world fact cannot be modelled by an attribute.

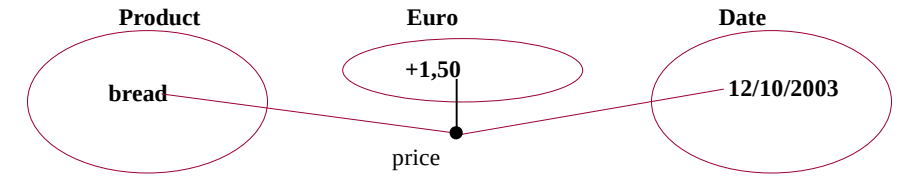


Figure 9 A ternary link between an object and data values

The method Catalysis [Dsou99, p.89] introduces the construct of parameterised attributes. Consequently, attaching a parameterised attribute such as `price(d:Date):Euro` to a class `Product` is allowed. We do not see a reason to introduce the construct of a parameterised attribute when the construct of a query is already available and it allows defining parameters. The introduction of the construct of a parameterised attribute would augment the problem of construct redundancy. Furthermore, Catalysis also allows introducing a parameterised attribute such as `price(s:Shop):Euro` attached to a class `Product`. This attribute represents a set of ternary links between shops, products and euros. We want to attach a model element, which represents a set of static properties, to *all* involved classes. Notice that in Catalysis the principle of no choice is violated: an analyst could also have decided to model a parameterised attribute such as `price(p:Product) :Euro` attached to a class `Shop`. The way to model a set of static properties in which more than one class is involved is discussed in chapter 4.

Although all methods for object-oriented analysis offer attributes as a means to model sets of static properties, we reject the construct of an attribute at the level of analysis for the following reasons:

- An attribute cannot represent a set of n-ary links between instances when n > 2.

- From our point of view, the construct of attributes is again too closely linked to design and programming. The construct of an attribute will be reserved for the design and implementation phases. Consequently, an attribute models data representation. In case of a vague border between analysis and design, possible confusion by the use of attributes is avoided.

2.2.4 Only Queries

The construct of a query is semantically richer than the construct of an attribute. Queries are more expressive than attributes, because queries can represent n-ary relations while attributes can only represent binary relations. Sets of n-ary links can be modelled by queries with parameters. Reconsider the example outlined in Figure 9. The query `price(date: Date): Euro` in Figure 10 represents the set of ternary links between products, dates and Euros.

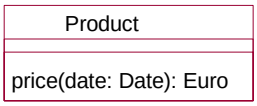


Figure 10 Queries can have parameters

The approach of using queries to model sets of links between data-values and objects reveals however several problems:

- A first problem is that we want to structure static properties around *all* involved classes. In the UML, a query can have a parameter that has a class as type. For example, a query such as `price(s:Shop):Euro` attached to a class `Product` is allowed. This query is not attached to the class `Shop`. This problem could be resolved by restricting the type of a parameter to a data type. As already stated, the way to model a set of

static properties in which more than one class is involved is discussed in chapter 4.

- Secondly, and more important, the principle of no choice is violated. An arbitrary choice must be made about which data type is specified as return type and which data types are specified as parameter types. Reconsider the example in Figure 9. A query `price(amount: Euro): Date [0..*]` attached to the class `Product` can also represent the set of ternary links between products, dates and Euros. Notice that links between instances must be modelled and not the functional requirements of the system. On top of the representation of real-world facts, functional requirements can be built. Assume the real-world fact that a product has a price on a certain date. The following examples of requirements can a specified based on this real-world fact: given a certain product and a certain date return the price of the given product at the given date and, as another example, given a certain product and a certain amount of money, return the set of dates on which the price of the given product equals the given amount. In the early stages of conceptual modelling, one must make abstraction of the functional requirements. In a conceptual model, real-world facts must be modelled at the first place and not functional requirements for the ultimate system. The modelling of functional requirements is further discussed in chapter 7. From our point of view, queries strongly suggest the representation of requirements. To stress the distinction between the representation of real-world facts and the representation of requirements, the construct of a query is abandoned for modelling static properties. Consequently, the principle of no choice is also no longer violated.

- To stress the distinction between modelling static properties and modelling functional requirements it is wise to abandon the notion of a “query” for modelling static properties.

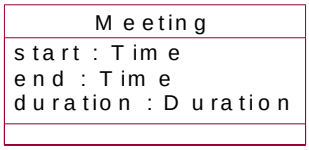
2.3 Patterns

2.3.1 Characteristics

The construct of a characteristic is introduced to model a set of links between objects and data values. A difference with the UML construct of an attribute is that the type of a characteristic can only be a data type and not a class. Another difference is that more than one data type is allowed as type for a characteristic. Consider as an example case three in Figure 11. The characteristic `price` has two data types, `Euro` and `Date`, as type. This characteristic represents the set of ternary links between accounts, dates and Euros. A difference with the UML construct of a query is that a characteristic cannot have parameters. The construct of a characteristic is also used to model a set of redundant static properties. Consider as an example the first two cases in Figure 11. The way to specify the redundancy relationship between the static properties is discussed in section 7.

Case one

Case two



Case three

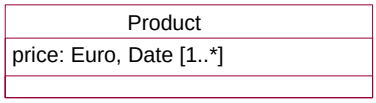


Figure 11 Using characteristics

This pattern is in line with the proposed features of languages for the development of information systems at the conceptual level [Borg85] where it is an important goal to allow the user to threat redundant and non-redundant information in a syntactically uniform manner in order to achieve further storage independence. This pattern also supports the extendibility of the conceptual model. Furthermore, due to the absence of attributes possible model alternatives and confusion of the analyst are avoided. In the next patterns, the characteristics will be modelled in the first compartment of the class. The second compartment of a class will be used to structure dynamic properties.

2.3.2 Implicit Queries of Characteristics

Given a characteristic `c` attached to a class `C`, an implicit query `c()` is given, which can be applied on an object of the class `C`. Such a query returns a set of data values that are linked with the object. At the level of analysis, queries will be used to specify restrictions, the effect of dynamic properties and the functional requirements. It is assumed that a query always returns a set, even if the multiplicity of the characteristic is `[1..1]` or `[0..1]`. When no data values are linked with the object, the query returns an empty set.

If a characteristic has more than one data type as type, then the implicit query of this characteristic returns a set of tuples. Further research is needed about the implicit queries of characteristics that have more than one data type as type. Further research is also needed about the usefulness of implicit queries that can be applied on data values and return a set of linked objects. In the method Catalysis [Dsou99, p.92], it is possible to navigate from a data value to a set of objects by the use of an implicit attribute.

3 Modelling Links between Objects

3.1 Problem Description

A static property cannot only be a link between objects and data values; it can also be a link between objects. Traditionally, associations are used to model these links. In exceptional cases, attributes are used to model links between objects. In this section, we discuss how links between objects can be specified in an adequate way. One of the questions to be answered is whether characteristics are an appropriate construct to model these links.

In section 1, it was decided to structure links between objects and data values around the involved class, instead of around the involved data type or around the involved data type and the involved class. For links between objects it would be a rather arbitrary decision to attach the model element to one of the involved classes. It would be even more difficult to guide the analyst in his choice for the most appropriate class. Consequently, static properties will be structured around *all* involved classes. No distinction between the involved classes is made.

In the example of Figure 12, links between accounts and persons and between accounts and banks can be observed. In order to model the first set of links, the model element will attached to the class `Person` and to the class `Account`. The second set of links will then be attached to the class `Account` and to the class `Bank`.

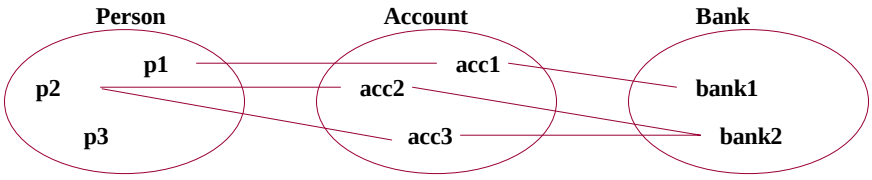


Figure 12 Links between objects

The UML Reference Manual [Rumb99, p.170] suggests to use attributes for links between objects and data values and to use associations for links

between objects. The manual arguments that for instances with identity, it is important to see the relationship in both directions. The argumentation for this decision is rather poor. Moreover, the analyst is free to attach attributes to both classes. Indeed, in the UML [Rumb99, p.168] an attribute can have a class as classifier. This seems to be in contradiction with the previous indication. The reason is that the UML doesn't make a strict distinction between analysis and design. In order to represent a possible design decision, the use of attributes that represent links between software objects cannot be excluded.

3.2 Anti-Patterns

3.2.1 Characteristics

Characteristics attached to all involved classes can be used to model a set of links between objects. Consider an example outlined in Figure 13. Characteristics are attached to the class `Bank` as well as to the class `Account`. Characteristics as such are not sufficient to model these links: the dependency between the characteristics must also be specified. This dependency can be modelled in the different ways. In the example, the dependency is specified as an invariant attached to the class `Account`. Three model elements are specified to represent a set of links between banks and accounts.

context Account **inv:** self.bank().account()→includes(self)

Figure 13 Using characteristics

The disadvantage of using characteristics to model links between objects is that at least three model elements are needed: at least two characteristics and a model element specifying the dependency between these characteristics. If a set of n-ary links where n > 2 must be modelled, then more than three model elements are necessary. The dispersion over more than one model

element of one simple real-world observation is disliked. Characteristics will not be used in order to prevent the dispersion of the specification over at least three model elements.

3.3 Patterns

3.3.1 Associations

An association attached to all involved classes is used to model a set of links between objects. No distinction is made between classes to model links between objects of those classes: the model element will be attached to all involved classes. Consider an example outlined in Figure 15. An association is attached to the class `Bank` and the class `Account`. This association represents the set of links between accounts and banks.

Figure 14 Using associations

If an association is used to represent a set of links between objects, only one model element is needed instead of at least three. An association will be used to model links between objects and it will be attached to all involved classes.

3.3.2 Implicit Queries and Implicit Invariants of Associations

Given the association in Figure 14, the implicit queries `bank()` and `account()` are given. The query `bank()` can be applied on a given account. This query returns the set of banks of the account. Due to the multiplicity the set has only one element. The query `account()` can be applied on a given bank. This query returns the set of accounts of the given bank. Given the association in Figure 14, the implicit invariant `context Account inv: self.bank().account()→includes(self)` is also defined. This

invariant specifies the dependency between the queries `bank()` and `account()`.

Assume that a subclass `SavingsAccount` of the class `Account` is added in the conceptual model of Figure 14. In such a case, an additional implicit query `savingsAccount()` is available to specify restrictions, the effect of dynamic properties and the functional requirements. The query `savingsAccount()` can be applied on a given bank and returns the set of saving accounts of the given bank.

Further research is needed about the given implicit queries and about the given implicit invariants in case of n-ary associations. Further research is also needed about the usefulness of implicit queries that can be applied on objects and return a set of *indirectly* linked objects. For example, assume that a class `Transaction` is added to the conceptual model of Figure 14 and that an association is defined between the class `Transaction` and the class `Account`. Is an implicit query `bank()`, which can be applied on a transaction, useful?

4 The Use of Primitive Data Types and Units

4.1 Problem description

In the UML and in programming languages, attributes heavily use predefined primitive data types such as `Integer`, `Real` and `String` [OMG01, p.6-29]. The UML does not impose any restriction on the use of these primitive data types. However, primitive data types are seldom needed in modeling real-world phenomena. They are often reflections of design and implementation decisions. Some analysis methods, for instance EROOS [Steeg01, p.74], impose restrictions on using primitive data types.

4.2 Anti-Patterns

During analysis, primitive data types can be easily misused. Consider an example in Figure 15. The data type of the characteristic `age` is `Integer`. In

our opinion, this is a pure design decision, reflecting the designer's choice how to represent ages. Another design decision could have been to represent ages as values of the primitive data type `Real`. Furthermore some information is missing if the data type `Integer` is used as type for the characteristic `age`. Is the age of a person expressed in years, in months, in days or in seconds? The decision how to represent the age of a person in a software object must be postponed.

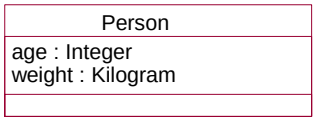


Figure 15 The misuse of primitive data types and units

Another problem related to the use of data types is also illustrated in Figure 15 by means of the characteristic `weight`. The data type of the characteristic `weight` is `Kilogram`. In our opinion, the decision concerning what unit will be used to represent the weight of a person is also a design decision. During analysis, it makes no sense to decide about the unit of weight, for instance kilogram or pounds.

4.3 Patterns

At the level of analysis, characteristics must reflect as good as possible real-world phenomena. Therefore, the data type of the characteristic `age` is `Duration` and the data type of characteristic `weight` is `Mass`. Notice that both data types make no statements about the data representation of both characteristics during design.

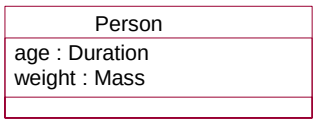


Figure 16 Avoiding primitive data types and units

The use of units and primitive data types is however not forbidden. Consider the conceptual model in Figure 17. The fact that the balance of an account is expressed in Euro is a clear business fact and not a decision that can be made by a designer during software engineering. The data type of the characteristic `balance` is a unit. The data type of the characteristic `numberOfTransactions` is `PositiveInteger`, which is a primitive data type. The characteristic reflects a quantity: the number of transactions of an account.

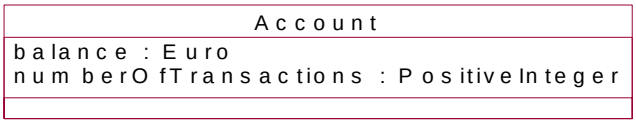


Figure 17 Using primitive data types and units

5 Restricting the Set of Data Values

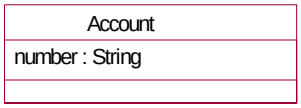
5.1 Problem Description

Specifying the data type of a characteristic restricts the possible set of data values with which an object can be linked. Consequently, specifying the data type defines an implicit constraint. Restricting the possible set of data values of a characteristic can be modelled in several ways. In this section, these different alternatives are evaluated.

The UML [Rumb99, p.248] suggests a rather primitive approach to introduce user-defined data types by enumerating it values or by referring to programming languages. Moreover, the constructs for introducing data types in the UML are vague.

5.2 Anti-Patterns

A first approach uses a more general data type and restricts the possible data values by means of an invariant attached to a class. Consider the example in Figure 18. Every account has a number, which is a string with specific restrictions. This is modelled as an invariant of the class Account.

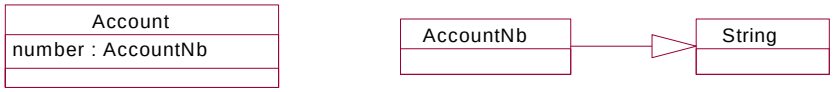


context Account **inv:** number() >= "0"
context Account **inv:** number() <= "999.999.999.999"

Figure 18 Restricting the set of data values

5.3 Patterns

Another approach is to define a new data type that subsets the more general data type. In the conceptual model of Figure 19, the data type AccountNb is introduced. This new data type is used as data type of the characteristic number.



context AccountNb **inv:** number() >= "0"
context AccountNb **inv:** number() <= "999.999.999.999"

Figure 19 Introducing sub data types

This second approach promotes reuse and adaptability. Reuse, because the new data types can be reused as the data type of an explicit argument of an event or as the data type of another characteristic. Adaptability, because when the specification of the data type must be changed, the specification must be modified only once.

Furthermore, new functions can be defined for the new data type. For instance, a new function `controlNb()` can be attached to the data type `AccountNb`. This function returns the last two numbers of the account number. In this way, this function can be reused every time the data type `AccountNb` is used. By the way, notice that the function `controlNb()` attached to the data type `AccountNb` is an example of modelling links between data values.

The result of analysis consists of two types of diagrams: a class diagram and a data type diagram. Notice that also this approach limits the use of primitive data types as `Integer`, `Real` and `String` in a class diagram.

6 Extending the Multiplicity of a Characteristic

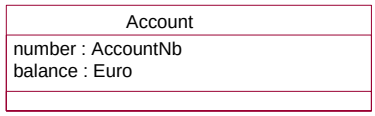
6.1 Problem Description

The multiplicity of a characteristic specifies the lower bound and the upper bound of the number of data values an object can be linked with at any given moment in time. In section 1, it was already pointed out that a consequence of attaching model elements to the involved classes was that a multiplicity was lost. Indeed, no specification is made about the lower bound and upper bound of the number of objects a data value can be linked with.

6.2 Anti-Patterns

It was implicitly assumed that a data value could be linked with an unlimited number of objects. This is illustrated in the conceptual model of Figure 20. An unlimited number of accounts can have the same balance. The default

multiplicity for the data type is therefore `[0..*]`. The question is how to specify this type of multiplicity in case the default is not valid? A first approach is to specify an explicit constraint. The invariant specifies that the number of account is unique for each account. In other words, an account number can be linked with at most one account.



context Account **inv:**
Account.AllInstances→forAll(acc1,acc2|acc1 <> acc2 implies
acc1.number() <> acc2.number())

Figure 20 The multiplicity of a characteristic

From our point of view, explicit defined constraints do not promote readability. Moreover, more general multiplicity constraints for data values are even more difficult to specify. For example, assume that an account number is linked with at least three accounts and with at most nine accounts.

6.3 Pattern

It is proposed to extend the notion of multiplicity of a characteristic by a second multiplicity range. The extended notion of multiplicity is illustrated in Figure 21. The multiplicity of the characteristic `number` is `[1,0..1]`. It means that every account has exactly one number and that this number is unique for each account. Furthermore, it is specified that not every valid account number must be linked with an existing account. The characteristic `balance` has the default multiplicity `[1,0..*]`. This means that every account has exactly one balance and that many accounts can have the same balance.

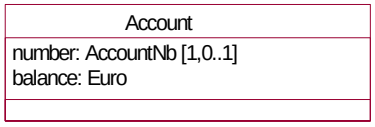


Figure 21 Adding a second multiplicity range

The introduction of a second multiplicity range for a characteristic avoids explicitly defined constraints. Further investigation is needed about the multiplicity for a characteristic that has more than one data type.

7 Modelling Dependencies between Static Properties

7.1 Problem Description

A ***dependency*** between static properties is defined as an everlasting relationship between static properties in which a change to static property may affect the static property. A dependency between static properties restricts the possible links of the involved instances. Information redundancy between static properties is a special kind of a dependency between static properties.

Consider two dependencies in Figure 22. The balance of an account must equal or exceed its limit all the time, and an account is said to be positive if and only if its balance is positive. In order to build conceptual models in a precise and complete way, existing dependencies between static properties must be represented in a formal way. The question is how to specify these dependencies in a formal and adequate way.

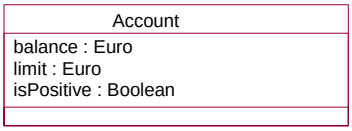


Figure 22 Dependencies between static properties

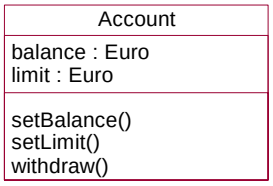
The UML offers the constructs of ***invariants***, ***preconditions*** and ***postconditions*** to formally specify dependencies between static properties [OMG01]. Which construct should be used? Are all these constructs needed at the level of analysis? Are those constructs sufficient? Recall [section 2] that derived attributes and explicit queries are not available anymore to represent dependencies. The constructs of invariants, preconditions and postconditions will be further analysed in detail in chapter 5 about modelling restrictions. In this section, these constructs are only judged on their adequateness to represent dependencies between static properties.

7.2 Anti-Patterns

7.2.1 Preconditions

In the OCL, a precondition is defined as a constraint that must hold for the invocation of the operation to which it applies [OMG01, p.2-33]. Preconditions can be used to model dependencies when the construct of a basic mutator is introduced in an object-oriented analysis method. More complex events are then obliged to use basic mutators to modify the state of the model. A precondition can be attached to a basic mutator. Then, more complex events derive their preconditions from the basic mutators, which are used in their specification. Can the introduction of the construct of a basic mutator be useful to represent dependencies (and restrictions in general)? Notice that this approach is frequently applied in object-oriented programming.

Consider the conceptual model in Figure 23. Two basic mutators are specified: `setBalance()` and `setLimit()`. The preconditions attached to both mutators are consequences of the dependency stating that the balance of an account may not drop below its limit. Notice that both preconditions are necessary. The more complex event `withdraw()` is specified with the help of the basic mutator `setBalance()`. The precondition of the event `withdraw()` is derived from the precondition of the basic mutator `setBalance()`.



context Account :: setBalance(amount : Euro)
pre: amount >= limit()
post: balance() = amount

context Account :: setLimit(amount : Euro)
pre: amount <= balance()
post: limit() = amount

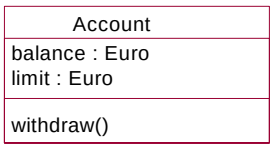
context Account :: withdraw(amount : Euro)
pre: balance() – amount >= limit()
post: setBalance(balance() – amount)

Figure 23 Preconditions modelling dependencies: basic mutators

A disadvantage of using preconditions is that the specification of the dependency gets dispersed over all basic mutators related to the static properties of the dependency. The dispersion of the specification of one real-world fact over several model elements is discouraged because it does not promote adaptability and readability. Moreover, the dispersion of the dependency is not restricted to the involved basic mutators. The specification of more complex events is also be influenced by the dependency. Furthermore, it is disliked that it is necessary to specify events in order to model a dependency between static properties. In order to respect the principle of abstraction, it should be possible to model these dependencies without representing dynamic properties.

When the construct of basic mutators is not supported, dependencies cannot be adequately modelled by preconditions. Indeed, in that case, nothing can prevent the introduction of additional events that could violate the dependency. Consider the example outlined in Figure 24. In this example, an event is specified directly to withdraw some amount of money from an account. The precondition reflects the dependency stating that the balance of an account may not drop below its limit. If this condition would only apply

to withdrawals, there would be no problem. However, if the dependency is of a more general nature, it might be violated by the introduction of other events, such as an event for changing the limit that applies to an account. Moreover, this approach would not prevent the dispersion of the specification of a dependency.



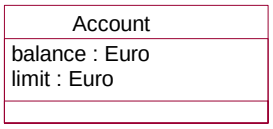
context Account :: withdraw(amount : Euro)
pre: balance() – amount >= limit()
post: balance() = balance()@pre – amount

Figure 24 Preconditions modelling dependencies: no basic mutators

7.2.2 Postconditions

7.2.2.1 Postconditions and Dependencies

In the OCL, a postcondition is defined as a constraint that must be true after the invocation of the operation to which it applies [OMG01, p.2-33]. In the UML, a query is an operation whose result can be specified with a postcondition. Consider the example in Figure 25. In alternative one, the specification of the dependency is dispersed over both involved queries. In the other alternatives, only one query is further specified to model the dependency. The choice between one of the involved queries is arbitrarily.



Alternative one
context Account :: balance()
post: result >= limit()

context Account :: limit()
post: result <= balance()

Alternative two
context Account :: balance()
post: result >= limit()

Alternative three
context Account :: limit()
post: result <= balance()

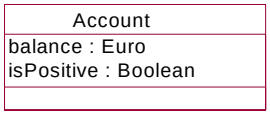
Figure 25 Postconditions modelling dependencies

Compared with the previous anti-pattern, no events must be defined; however, the disadvantage that the specification of the dependency is dispersed over several model elements is, in general, also true for postconditions. To avoid the dispersion, the analyst can also decide to pick out only one query to specify the postcondition. This is not an ideal solution because the choice of a query is, in general, arbitrarily.

7.2.2.2 Postconditions and derived Queries

In the UML [OMG01, 2-44], a model element is defined as derived if it can be computed in terms of other model elements. We define a derived query as a query, which can be computed in terms of other queries. In the example outlined in Figure 22, the query `isPositive()` is derived.

One could decide to attach the postcondition, which specifies the dependency, to a derived query. Consider the example in Figure 26. The query `isPositive()` is further specified with a postcondition. This postcondition specifies the dependency. Notice that the expression `(balance() >= 0)` evaluates to true or false. In this case, the disadvantages of using postconditions to model the dependency disappear: the specification of the dependency is not dispersed among several model elements and the choice of the query is not arbitrarily.



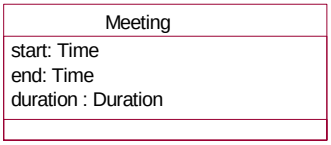
context Account :: isPositive()
post: result = (balance() >= 0)

Figure 26 Derived queries and postconditions

One could decide to model dependencies, in which derived queries are involved, with the help of postconditions attached to a derived query. From our point of view, this approach has several disadvantages. These disadvantages are discussed in the next paragraphs:

- A first disadvantage is that in a dependency more than one derived query can be present. Consequently, the selection of a query to specify the dependency is arbitrarily. In Figure 27, two examples are given. In each case, a dependency is indicated and all queries are derived. Consider case one. Three possibilities exist to model the dependency. During analysis, the choice between the possibilities is arbitrarily. Consider case two. During design, it is logic to calculate the age of a person on the basis of his birth date. This prevents that the age of a person must be constantly updated. This design concern does not exist during analysis. During analysis, the choice to attach the postcondition to the query `age()` or to the query `birthDate()` is arbitrarily.

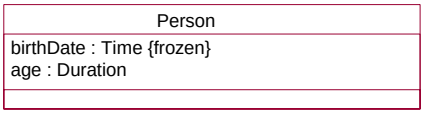
Case one



-- Dependency: end() = start() + duration()

alternative one: **context** Meeting duration() **post:** result = end() – start()
alternative two: **context** Meeting end() **post:** result = start() + duration()
alternative three: **context** Meeting start() **post:** result = start() + duration()

Case two



-- Dependency: age() = now() – birthDate()

alternative one: **context** Person :: birthDate() **post:** result = now() – age()
alternative two: **context** Person :: age() **post:** result = now() – birthdate()

Figure 27 More than one derived query in a dependency

- A second disadvantage is that a distinction between queries with postconditions and queries without postconditions is made. This is close to distinguishing between attributes and queries. From our point of view, this is another way of suggesting data representation. Like we rejected the possibility to model case one in Figure 27 with two attributes and a query, we want to reject the possibility to model this case with two queries and a derived query.

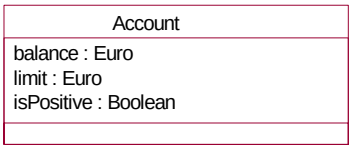
- Thirdly, no answer is given about how to model dependencies, in which no derived queries are involved. In that case, or in the case more than one derived query is involved in a dependency, one could propose to model dependencies with invariants. From our point of view, a rather artificial distinction is made between dependencies with one derived query involved and with no or more than one derived query involved. At the level of static properties and their dependencies, there is no semantic need to introduce two constructs to model dependencies.

- A fourth disadvantage is that the construct of a derived model element is only introduced for technical modelling reasons. This construct helps the analyst to determine the queries to which the postconditions could be attached. The construct of a derived query was not introduced to express real-world semantics. One could argue that a derived model element specifies that a static property can be computed in terms of other static properties. This is correct, but normally this computational fact must not be represented. For instance, the fact that one can compute an amount of a transaction of an account in terms of the other amounts of the transactions of this account and the balance of this account must not be

modelled. Normally, an analyst wants to model dependencies between static properties, he does not want to represent that a static property can be computed in terms of other static properties. In other words, in order to represent real-world facts as static properties and its dependencies it is not necessary to determine if a query is derived or not. The determination of a query as derived in a dependency does in no way limit the possible state of the model. It is the dependency itself that limits the possible state of the model. Other constructs as characteristics, classes, associations and multiplicities were introduced to express real-world semantics.

7.3 Patterns

In the UML, an invariant attached to a class is defined as a constraint that must be true for all objects of that class at any time [OMG01, p.6-5]. A dependency between static properties is true at all times. Consequently, an invariant can be used to model a dependency. Consider the conceptual model in Figure 28. In this model, the dependencies are modelled with the help of invariants. The dependencies, which are present in the example of Figure 27, are also modelled by means of an invariant.



context Account **inv:** balance() >= limit()
context Account **inv:** (balance() >= 0 Euro) = isPositive()

Figure 28 Invariants modelling dependencies

When an invariant is used to model the dependency, the specification is not dispersed over several model elements and no arbitrary choices between queries must be made. Important to notice is that the use of invariants should be further evaluated in relation with the specification of dynamic properties and that the proposed solution to model dependencies with invariants is not

final. The use of invariants in combination with the specification of dynamic properties is further discussed in the chapter 5 about specifying restrictions.

8 Partial Characteristics

8.1 Problem Description

After the decision concerning how to model static properties and dependencies between properties a next question arises: how to assign characteristics and associations to the correct class? This section about partial characteristics and the sections 9 and 10 will help the analyst to answer this question.

A partial characteristic is defined as a characteristic that has a minimum multiplicity of zero. In this section, the use of partial characteristics will be evaluated. In the literature [Asse96, p.6] [Stee00, p.63], many authors warn against or forbid the use of partial attributes in conceptual modelling. Partial attributes could lead to problems with understanding the meaning of the attributes. No empirical evidence is however found to prove that users of conceptual models with mandatory attributes better understand the real-world domain being modelled [Boda98].

In the UML, an attribute can have a multiplicity `[0..m]`. Such a multiplicity implies the possibility of a null value [OMG01, p.3-43]. In [Elma00, p.473], it is argued that null values can have multiple interpretations: the attribute does not apply (yet), the value for the attribute is unknown (and therefore not recorded) or the value is known but has not been recorded yet. A single representation for null values compromises the different meanings it may have. Notice that the last two interpretations are impossible at the level of analysis: the storage of information is not an issue during analysis. We consider the construct of a null value as a design or implementation construct.

From our point of view, the multiplicity `[0..m]` of a characteristic indicates the possibility that the implicit query can return an empty set. Just like an association end with a multiplicity of `[0..m]` indicates that the implicit

query returns a set that can be empty). As an example, consider a class `Person` with a partial characteristic `dateOfMarriage : Date [0..1]`. If the query `dateOfMarriage()` is returning an empty set for a given person, then the given person has currently no date of marriage. In other words, he or she is not married. In the same way, a query `account()` applied to a given person and returning an empty set, reflects the fact that the given person has currently no accounts.

8.2 Anti-Patterns

8.2.1 Violating the Principle of Completeness and Abstraction

A possible approach is to forbid the use of partial characteristics. Consider the example outlined in Figure 29. Assume that a person has always just one name and that a person can have at most one salary. In the example, a subclass `Employee` is introduced to avoid partial characteristics. The introduction of the class `Employment` instead of the class `Employee` could also be considered to avoid the partial characteristic, but the discussion is here not about the use of generalization or delegation.

Figure 29 Violating the principle of completeness and abstraction

The principle of abstraction is violated. The analyst is forced to search for a set of objects in which all elements have always a salary. More real-world facts than necessary are represented: employees have always a salary, employees are persons and employees have always a name. This violation of the principle of abstraction leads to a more complex model.

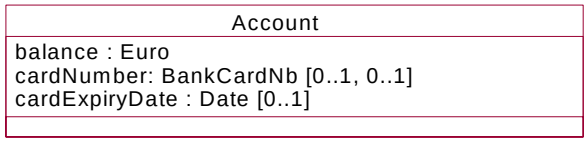
The principle of completeness is also violated: some real-world facts are not represented. In the example, the fact that a person can have maximum one

salary is not represented. An employee has just one salary. No statements are made about the number of salaries a person can have.

Notice that the conceptual model is not inadequate as such. Assume the following observed real-world facts: persons have always just one name, employees are persons and employees have always just one salary. If these real-world facts must be modelled, the conceptual model of Figure 29 is considered to be appropriate.

8.2.2 Explicit Invariants testing the Number of Elements in a Set

The unlimited use of partial characteristics can lead to an exhaustive list of complex and explicit invariants, which test if implicit queries are returning empty sets. Consider the example in Figure 30. Assume that an account can have a card number and a card expiry date. If an account has a card number it has also a card expiry date. This restriction is specified in the invariant. An analogue example could be made with characteristics such as `accountBalances` and `accountNumbers` attached to the class `Person` and an invariant stating that the number of elements in both sets must be the same.



context Account **inv:** cardNumber()→size()= cardExpiryDate→size()

Figure 30 Explicit invariants testing the number of elements in a set

As it will be discussed in chapter 6 about restrictions, explicitly specified constraints must be avoided as much as possible. From our point of view, the absence of explicit constraints promotes the readability of the model.

8.3 Pattern

8.3.1 Use Partial Characteristics

It is sometimes argued that the use of partial characteristics increases the heterogeneity of classes while homogeneous classes are considered to be a quality factor [Asse96, p.279]. Homogeneous classes do not contain highly dissimilar objects. Consider the conceptual model outlined in Figure 31. The partial characteristic `salary` is attached to the class Person.

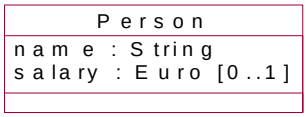


Figure 31 Using partial characteristics

The class `Person` in the conceptual model of Figure 31 is as heterogeneous or homogeneous as in the conceptual model of Figure 29. This class represents namely the same set of objects. Furthermore, the use of partial characteristics promotes the principle of abstraction and completeness.

8.3.2 Avoid explicit Invariants testing the Number of Elements of a Set

Analysts must avoid the specification of explicit invariants, which test the number of elements in a set. Consider the conceptual model in Figure 32. Compared with the conceptual model of Figure 30, no explicitly specified invariants are present.

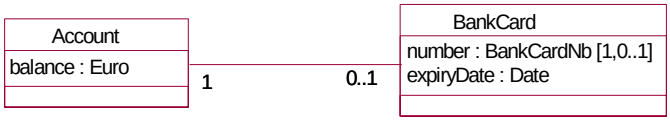


Figure 32 No explicit invariants testing the number of elements of a set

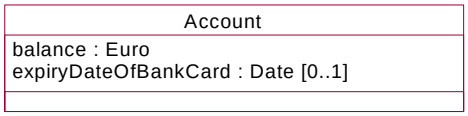
As stated in [Vanb94, p.190], the absence of explicitly specified constraints leads to better-structured models. Avoiding explicitly specified constraints promotes the principle of extendibility but violates the principle of abstraction. This will be further discussed in chapter 5 about modelling restrictions.

If relevant, partial characteristics such as `cardExpiryDate` and `cardNumber` could be attached to the class `Account`. In this example, partial characteristics allow to model information redundancy. In addition to both partial characteristics, two invariants must be added to the model. These invariants must specify the information redundancy relationship between the characteristics `cardExpiryDate` and `expirydate` and the characteristics `cardNumber` and `number`. Note that explicitly defined invariants that represent information redundancy relationships are allowed.

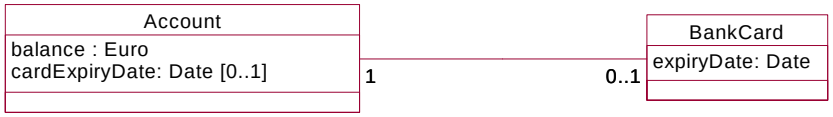
8.3.3 The Principle of Extendibility

Partial characteristics promote the principle of extendibility. Consider the conceptual models of Figure 33. The second and the third model are extensions of the first model. These extensions are examples where the principle of extension is satisfied: no modification of the existing specification of first initial model is needed.

Model one: intial model



Model two: extended model using delegation



context Account **inv**: cardExpiryDate () = bankCard().expiryDate()

Model three: extended model using generalization

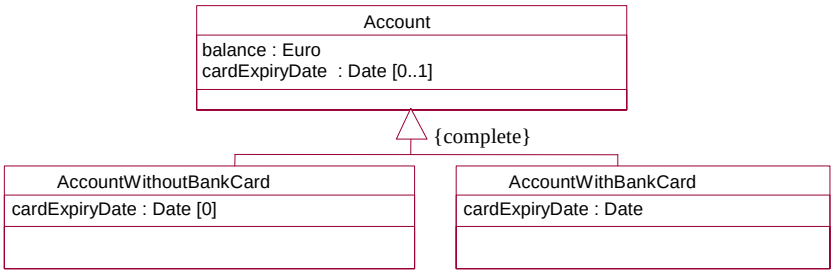


Figure 33 The principle of extendibility

Reconsider model three. The class `Account` has a partial characteristic `cardExpiryDate`. The multiplicities of characteristics are in fact constraints. As it will be discussed in the chapter 6, constraints can be strengthened at the level of specialization. Consequently, the minimum multiplicity can be increased at the level of specialization. This is illustrated in the characteristic `cardExpiryDate` of the class `AccountWithBankcard`. In the same way, the maximum multiplicity can be decreased at the level of specialization. This is illustrated in the characteristic `cardExpiryDate` of the class `AccountWithoutBankCard`.

The multiplicity `[0]` of the characteristic `cardExpiryDate` in the class `AccountWithoutBankCard` indicates that the static property does not exist for objects of this class. The specification of multiplicity `[0]` for a characteristic is not equivalent with not modelling the characteristic. Two reasons exist for not modelling a characteristic: the characteristic is irrelevant or the characteristic does not exist. For example, the multiplicity `[0]` of a characteristic `price` in a class `Person` is necessary if the analyst wants to represent that persons cannot have a “price”. The weight of a person is not modelled in Figure 31 because it is irrelevant.

Reconsider the specification of the invariant in the second model. It is assumed that a query applied to an empty set yields an empty set or more formally `{}.q()={}`. In the OCL [Warm03, p.82], the result of the expression `{}.q()` is undefined.

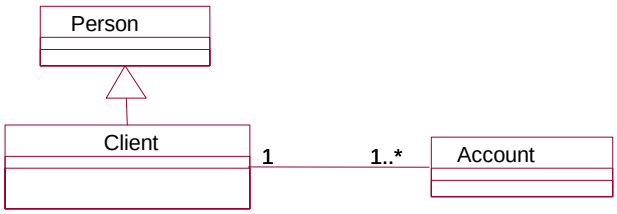
8.3.4 Partial Association Ends

It is surprisingly to observe that the use of partial characteristics is such a contentious issue in conceptual modelling, while the use of associations, which have at least one association end with a minimum multiplicity of zero, seems to pose no problem. While methodologists often advocate the transformation of the conceptual model of Figure 31 into the conceptual model of Figure 29, a transformation in Figure 34 of the first model into the second model is almost never promoted.

Model one

*-- **context** Person :: account(): Account [0..*]*

Model two



*-- **context** AccountHolder::account(): Account [1..*]*

Figure 34 Partial association ends

8.3.5 Conclusion

Although many researchers advocate against the use of partial characteristics, we are in favour of it. Partial characteristics satisfy the principle of abstraction, completeness and extendibility and allow modelling information redundancy. Forbidding partial characteristics doesn't promote comprehensibility or the homogeneity of classes. Avoiding explicit invariants testing the number of elements of a set helps the analyst to detect

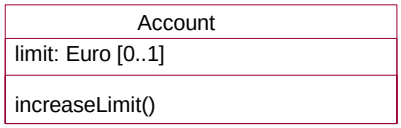
the correct class for a given characteristic leading to better-structured models.

9 Avoiding Non-Definedness

9.1 Problem Description

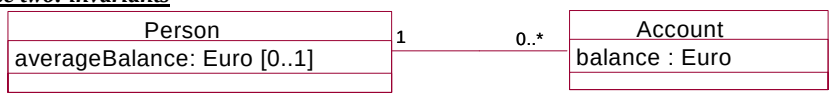
Queries are used to specify invariants and events. If a query can return an empty set, the invariant or event specified with the help of such a query, may not be fully defined. Consider the first example outlined in Figure 35. Because an account has not always a limit, the characteristic `limit` is partial in the class `Account`. Assume that the limit of an account can be increased, reflected by the definition of the mutator `increaseLimit()`. The mutator is not well defined in case the account has no limit: the query `limit()@pre` is used in the specification of the mutator and returns an empty set in case the given account has no limit. The expression `{} + amount` results in an undefined state for the characteristic `limit`. Consider the second example in Figure 35. The invariant is inaccurately defined. In case a person has no accounts, the invariant is not well defined due to a division by zero. Undefined or partially undefined specifications must be avoided. How can this non-definedness be avoided in the specification of invariants and events?

Case one: events



-- badly defined mutator
*-- **context** Account :: increaseLimit(amount : Euro)*
*-- **post:** limit() = limit()@pre + amount*

Case two: invariants



-- badly defined invariant
*-- **context** Person **inv:** averageBalance() = account().balance()→sum() / (account()→size())*

Figure 35 Non-definedness

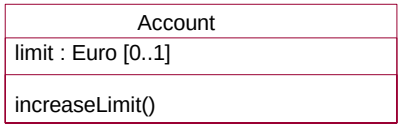
Note that according to the OCL [Warm99, p.95], the query `sum()` returns 0 when the query is applied on an empty set, e.g. if a person has no accounts. We are not in favour of this assumption because it is preferred to make a distinction between 0 and an empty set. For instance, assume a characteristic `totalBalance: Euro [0..1]` attached to the class `Person` in Figure 35. In addition, the model is extended with an invariant `totalBalance() = account().balance()→sum()` attached to class `Person`. If a person has no accounts, the query `totalBalance()` should return an empty set. If a person has accounts, the totalBalance could equal 0 Euro. We assume that a query, e.g. the query `sum()`, applied on an empty set returns an empty set. As a result a distinction can be made between persons that have no accounts and persons that have accounts, but where the total balance of their accounts equals 0 Euro.

9.2 Anti-Patterns

9.2.1 More complex Specifications

Non-definedness can be avoided by the introduction of more complex specifications, for instance with the help of the if-then-else construct. Consider the examples in Figure 36. In both cases, the condition in the if-then-else construct tests if a query returns an empty set. In the first case, a precondition could be used instead of the if-then-else construct, but the discussion is here not about preconditions versus if-then-else expressions.

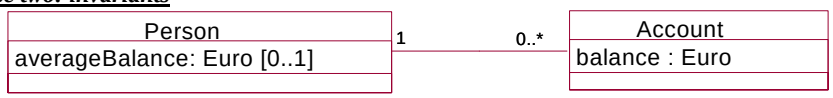
Case one: events



context Account :: increaseLimit(amount : Euro)

post: **if** limit()@pre→isEmpty()
then limit()→isEmpty()
else limit() = limit()@pre + amount
endif

Case two: invariants



context Person **inv:**
if account()→isEmpty()
then averageBalance()→isEmpty()
else averageBalance() = account().balance()→sum() / account()→size()
endif

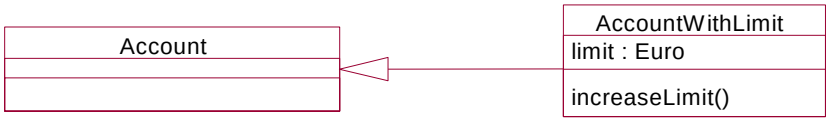
Figure 36 Complex specificiatons

In object-orientation, if-then-else constructs should be avoided. Generally, more complex specifications must be avoided as much as possible because they do not promote readability.

9.2.2 Avoiding Queries that can return empty Sets

Another approach to avoid non-definedness in the specification of invariants and events is to avoid implicit queries that can return empty sets. In this approach, no constraints, which test when queries return empty sets, can be used. Consider the examples outlined in Figure 37. Partial characteristics and partial association ends are avoided by adding new classes. Notice that instead of generalization, delegation can be used to avoid partial characteristics and partial association ends, but the discussion is here not about delegation versus generalization.

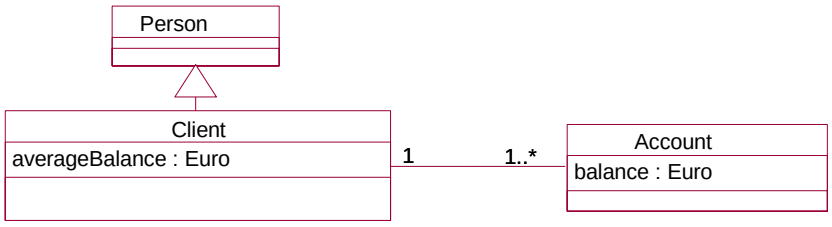
Case one: events



context FrozenAccount :: prolongFrozen(duration : Duration)

post: frozenUntil() = frozenUntil()@pre + duration

Case two



context Client **inv:**
averageBalance() = account().balance().sum() / account().size()

Figure 37 No queries that can return empty sets

The disadvantages of this approach, such as more complex models and the violation of the principle of completeness and abstraction, are already discussed in section 8.

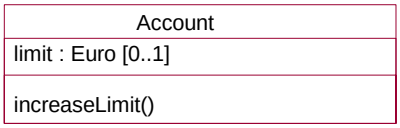
9.3 Patterns

From our point of view, the non-definedness can be caused by the difference that is made between instances and sets of instances. For instance, the addition of two integers is defined but not a set of integers plus an integer. If it is assumed that an implicit query always returns a set (some sets containing only one element) and that queries can be applied to sets, then the non-definedness can be avoided. An expression `S.q()` where `q()` is a query and `S` a set returns a set where the query was applied on each element of the set. For instance, a query `plus(3)` applied on a set `{5,8}` returns the set `{8,11}`. The application is denoted by a dot: `{5,8}.plus(3)`. A query applied on an empty set, returns an empty set. An expression `S→q()` where `q()` is a query and `S` a set indicates that the query is applied on a set and not an each element of the set, e.g. the expression `S→size()` returns the number of elements of the set `S`.

Consider the first case in Figure 38. It is supposed that the query `limit()@pre` returns a set with one data value of the data type `Euro` or an

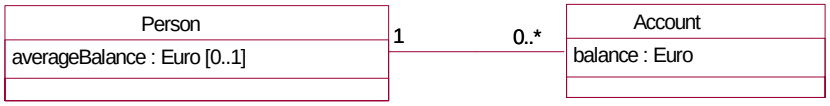
empty set. Furthermore, it is assumed that a query `plus(amount: Euro): Euro` is defined in the data type Euro. This query `plus()` is applied on a set with one element or on an empty set. If it was applied on an empty set, the query `plus()` returns an empty set. An analogous reasoning could be made for he second case. Consequently, the expression `{}.divideBy(0)` results in an empty set.

Case one: events



context Account :: limit(amount : Euro)
post: limit() = limit()@pre.plus(amount)

Case two: invariants



context Account **inv:**
averageBalance()=account().balance()→sum().divideBy(account()→size())

Figure 38 Avoiding non-definedness: events and invariants

This approach promotes uniformness. A query always returns a set of instances. No distinction is made between queries returning one instance, and queries returning sets of instances. The approach also promotes adaptability: the multiplicity of a characteristic or association end can change without the necessity to change existing specifications. We are in favour of this last alternative to avoid non-definedness because it supports abstraction. It is also possible to reason about the average balance of all persons while in the second anti-pattern it was only possible to reason about the average balance of clients. Notice that this approach needs further investigation. For example, what are the results of the expressions `{3}.plus({})` and `{3,4}.plus({5,6})`?

10 The Third Normal Form in Object-Oriented Analysis

10.1 Problem Description

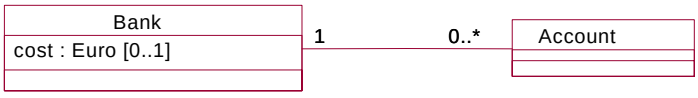
The ***third normal form*** (3NF) is a key construct in relational database design [Elma00]. The 3NF is a step in the normalization process. This process avoids the plural storage of the same information and update anomalies in databases. In this section, the possible influence of this form on object-oriented analysis is evaluated. Can this form help to determine the correct class of a characteristic? Consider the example in Figure 39. Assume that each account has a cost during its whole lifecycle and that the cost is the same for accounts at the same bank. What is the appropriate class for the characteristic `cost`?

Figure 39 What is the approriate class for the characteristic cost?

10.2 Anti-Patterns

10.2.1 Explicit Invariants testing the Number of Elements in a Set

In a first approach, a partial characteristic `cost` is attached to the class Bank: a bank must not have a cost when it has no accounts. The fact that an account has always a cost is modelled by an invariant. In section 8, we have seen that explicit constraints testing the number of elements in sets indicate a badly structured model.

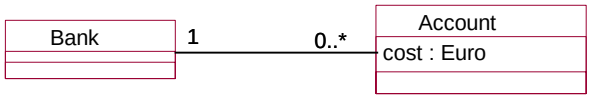


context Account **inv:** bank().cost()→size()=1

Figure 40 Explicit invariants testing the number of elements in a set

10.2.2 Explicit Invariants representing Transitive Dependency

Avoiding explicit invariants testing the number of elements in a set is not sufficient to determine the class for a characteristic. Consider the example in Figure 41. The characteristic `cost` is moved to the class `Account`. The invariant specifies that each account at the same bank has the same cost.



context Bank **inv:** account()→forAll(acc1,acc2|acc1.cost()=acc2.cost())

Figure 41 Explicit invariants representing transitive dependency

If this approach would not be rejected, every characteristic of a bank can be modelled in the class `Account`, adding the same kind of invariant. For instance, the characteristic `bankName` can then also be modelled in the class `Account`. Another disadvantage is that a possible mutator for the cost is not so easy to specify. Indeed, the mutator must change the cost for all accounts at the same bank.

In relational database design, the cost of an account will also not be modelled in the relation `Account`. Such a design would be rejected because of the third normal form. The third normal form is based on the concept of ***transitive dependency***, which in turn is based on the concept of ***functional dependency***. For precise definitions of relational databases, the normal forms, functional dependency and transitive dependency we refer to the literature about database design [Elma00].

In database design, the third normal form rejects all transitive dependencies between attributes and the primary key. In object-oriented analysis, the implicit identity of an object can act as the primary key, and queries can act as attributes. The resemblance between transitive dependency in database design and in object-oriented analysis is illustrated in Figure 42.

Transitive Dependency in Database Design
Account (accountnr, cost, banknr)
accountnr ⇒ banknr
banknr ⇒ cost
not (banknr ⇒ accountnr)

Transitive Dependency in Object-Oriented Analysis
context Account
self ⇒ bank()
bank() ⇒ cost()
not (bank() ⇒ self)

Figure 42 Transitive dependency in design and analysis

In database design, a functional dependency from X to Y is denoted as X ⇒ Y. It is said that Y is functional dependent upon X. If (X ⇒ Y) and (Y ⇒ Z) and (not (Y ⇒ X)) then Z is said to be transitive dependent from X. The relation `Account` has three attributes `accountnr`, `cost` and `banknr`. The first attribute is the primary key. The attribute `banknr` is functional dependent on `accountnr`, while the attribute `cost` is functional dependent on `banknr`. The attribute `banknr` is not a candidate key. Consequently, the attribute `cost` is transitive dependent on `accountnr`.

In object-oriented analysis, the concept of transitive dependency is illustrated with the help of the implicit identity of the object, denoted by `self`. In the bottom half of Figure 42, the query `cost()` is transitive dependent upon the object (or upon the implicit identity) via the query `bank()`. This transitive dependency is disallowed and is reflected in the invariant in Figure 41.

10.3 Patterns

10.3.1 The 3NF in Analysis

In object-oriented analysis, the third normal form helps to avoid explicitly specified invariants that represent transitive dependency and helps to detect

the appropriate class for a model element that represent a set of static properties. Consider the conceptual model in Figure 43. A new class `AccountOffer` is added to the model. An object of the class `AccountOffer` is created when the bank decides to offer accounts to its clients. At that moment, the bank decides about the cost of an account. Specifying a mutator that modifies the cost poses no particular problems.

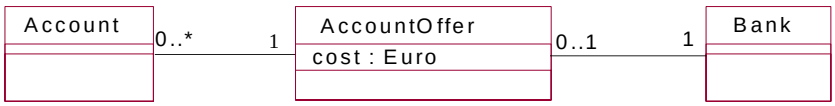
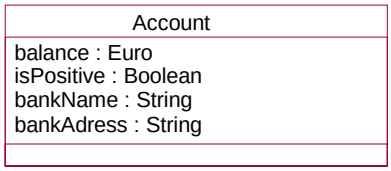


Figure 43 The third normal form in object-oriented analysis

10.3.2 Refinement of the 3NF in Object-Oriented Analysis

Transitive dependencies cannot always be avoided in object-oriented analysis because it is allowed to model information redundancy, while during relational database design it is assumed that information redundancy has already been eliminated. Consider the first alternative example in Figure 44. The query `isPositive()` is transitive dependent on the implicit identity via de query `balance()`. This constraint is represented by the first invariant and cannot be avoided because the characteristic `isPositve` is a model element that represents information redundancy.

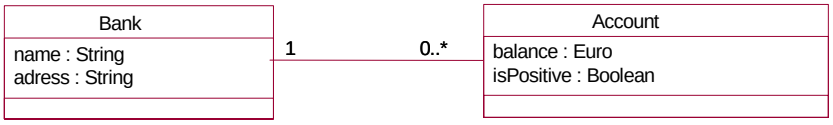
Alternative one



context Account **inv:** isPositive() = (balance() >= 0 Euro)

context Account **inv:**
Account.allInstances→forAll(acc1,acc2|acc1.bankName() = acc2.bankName() implies acc1.bankAdress() = acc2.bankAdress())

Alternative two



context Account **inv:** isPositive() = (balance() >= 0 Euro)

Figure 44 Transitive dependency cannot always be avoided

The query `bankAdress()` is transitive dependent on the implicit identity via the query `bankName()`. This constraint is represented by the second invariant. In the second alternative, this transitive dependency is avoided by introducing a class `Bank` and by moving both characteristics to this class. One could argue that only the characteristic `bankAdress` must be moved to the class `Bank` in order to respect the third normal form in object-oriented analysis. This is not the correct because, if only the characteristic `adressBank` is moved to the class `Bank`, the query `bankName()` is transitive dependent on the implicit identity via the query `bank()`.

Notice that the reasons to avoid transitive dependencies in relational database design and in object-oriented analysis are strongly different. In relational database design, the reasons are efficiency in space (avoid the data storage of the same information more than once), and efficiency in time (avoid update anomalies). In object-oriented analysis, the reason to avoid transitive dependencies is the search for a more stable, more comprehensive and better-structured conceptual model.

10.3.3 Conclusion

The analyst must avoid explicitly modelled constraints caused by transitive dependencies, as far as they are not caused by model elements representing redundant information. These transitive dependencies can be avoided by attaching the characteristic to the correct class. In some cases, this may lead to the introduction of additional classes in the conceptual model.

Once a set of static properties has been observed in the external world, strict criteria are needed, unambiguously indicating how the given properties are to be incorporated in the conceptual model in the most appropriate way. In the sections 8, 9 and 10, we have introduced criteria for attaching characteristics to classes. It cannot be overstressed that these criteria are only suited for building conceptual models. They could be violated in design models due to a trade-off in software quality factors. For example, the model in Figure 40 could be chosen as a solution during the design phase.

11 Other Normal Forms in Object-Oriented Analysis

11.1 Problem Description

In this section, the possible influence of the first normal form and the second normal form in relational database design on object-oriented analysis is briefly described. In order to be complete, further research is needed on the relationship between object-oriented analysis and the other normal forms, in particular the Boyce-Codd normal form, the fourth normal form and the fifth normal form.

11.2 Patterns

11.2.1 The First Normal Form

The first normal form in database design disallows multivalued attributes caused by the need of a fixed record length in relational database design. In object-oriented analysis, this poses no problem: multi-valued characteristics are allowed. Consider the example in Figure 45. A person can have different first names.

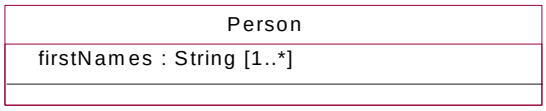


Figure 45 The first normal form in object-oriented analysis

The first normal form in design also disallows composed attributes. In section 2, it was stated that characteristics can have more than one data type as type.

11.2.2 The Second Normal Form

The second normal form in database design prevents the use of too large composed primary keys. Primary keys for tables in relational databases must be as small as possible. Because in object-oriented analysis, one is working with the implicit identity of an object instead of a primary key, the second normal form is irrelevant in object-oriented analysis. However, one can imagine cases where a combination of characteristics is unique for each object of a class.

Consider the first alternative in Figure 46. A class `Granteeship` is introduced to model all granteeships of accounts. An account can have different persons as grantee. The combination of the account number and the social security number of a grantee is unique for each granteeship, supposing that the same person can never be more than once a grantee for the same account. This restriction is modelled in the first invariant.

In addition, it is possible that another characteristic of the class is functional dependent on one of the queries of the unique combination of queries. In database design, such cases are not resolved by the third normal form, while in object-oriented analysis these cases all lead to a transitive dependency due to the presence of the implicit identity of an object. Indeed, the query `balance()` is transitive dependent on the implicit identity via the query `accountNumber()`. The third normal form disallows this transitive dependency. Consequently, no other criteria are needed to prevent such cases in object-oriented analysis. This restriction is modelled in the second invariant. Notice that the characteristic `accountNumber` is not unique in the class `GranteeShip`. The transitive dependency is avoided in the second alternative.

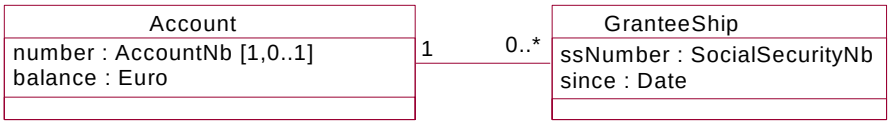
Alternative One

context GranteeShip **inv:**
GranteeShip.allInstances→forAll(g1,g2| g1 < > g2 implies
(g1.accountNumber() < > g2.accountNumber() or
g1.granteeSsNumber() < > g2.granteeSsNumber()))

context GanteeShip **inv:**
GranteeShip.allInstances→forAll(g1,g2| g1.accountNumber() = g2.accountNumber() implies
g1.balance() = g2.balance())

--Transitive Dependency
--self ⇒ accountNumber()
--accountNumber()⇒ balance()
--not (accountNumber()⇒ self)

Alternative Two



context Account **inv:**
granteeShip()→forAll(g1,g2| g1 < > g2 implies g1.ssNumber() < > g2.ssNumber())

Figure 46 The second normal form

11.2.3 Conclusion

During analysis, the first normal form and the second normal form do not influence the way characteristics are modelled and attached to classes.

12 Modelling Dependencies on the Model Layer

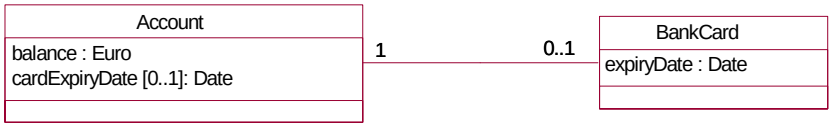
12.1 Problem Description

In this section and in section 13, two ideas will be discussed that can improve the adaptability of conceptual models. In [Stee02, p.42], adaptability is defined as a quality factor for software systems. This quality factor measures the ease to change existing software systems. At the level of analysis, this quality factor measures the ease to change an existing conceptual model. In this section, the idea was to specify not only dependencies on the user object level, but also on the model level. In section 13, the idea was about encapsulation at the level of analysis. These ideas are applied to model elements that represent sets of static properties. The application makes the ideas more concrete. These ideas could be also applied to other model elements. In both sections, the focus was on the ideas to improve the adaptability of conceptual models and certainly not on the way these ideas must be worked out. Both ideas seem to be promising, but need further investigation.

The UML [OMG01, p.2-4] [Klep03, p.85] [Fran03, p.105] has a four-layer metamodel architecture. The four different layers are: ***the user object layer***, ***the model layer***, the metamodel layer and the meta-metamodel layer. The primary responsibility of the model layer is to define a language that describes a problem domain. The model layer contains models, e.g. the class diagram of Figure 47. Elements of this model are the class `Account` and `BankCard`, the characteristics of these classes, the association between these classes and the invariant. The user object layer contains the instances of the model. The other layers fall outside the scope of this section.

In section 7, the discussion was focused on how to model dependencies between static properties. These dependencies are situated at the object layer. Consider the example in Figure 47. There exists a dependency between the partial characteristic `cardExpiryDate` in the class `Account` and the characteristic `expiryDate` in the class `BankCard`. This dependency is expressed in the invariant. There may also exist dependencies at the model layer. For instance, dependencies may exist between the multiplicities and data types of different characteristics. In the example, there exist a

dependency between the multiplicity of the characteristic `cardExpiryDate`, the multiplicity of the characteristic `expiryDate` and the multiplicity of the association end `bankCard`. Furthermore, there exists a dependency between the data type of the characteristic `cardExpiryDate` and the data type of the characteristic `expiryDate`.



context Account **inv:** cardExpiryDate() = bankCard().expiryDate()

Figure 47 No specification of dependencies on the model layer

Let us assume that the real-world situation changes, and that an account can be linked with several bankcards. In such a case, not only the multiplicity `[0..1]` of the association end `bankcard` must be modified into `[0..*]`, but also the multiplicity of the characteristic `cardExpiryDate`. It is obvious that the conceptual model of Figure 47 is not very adaptable, because specifications at several points in the conceptual model must be changed. In order to better support adaptability, these dependencies must be expressed. How can dependencies, which are situated at the model level, be specified?

12.2 Anti-Patterns

The conceptual model in Figure 47 is an anti-pattern: the dependencies at the model layer are not specified.

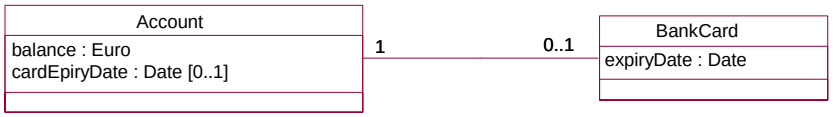
12.3 Patterns

12.3.1 Introducing Model-Layer Queries

In the OCL [OMG01, p.6-29], some queries that give access to the model layer are already predefined. For instance, the queries `attributes()` and

`operations()` return respectively the set of names of attributes and the set of names of operations of a certain classifier.

We suggest introducing some additional predefined model-layer queries as `minMultiplicity()`, `maxMultiplicity()` and `DataType()`. These queries can be applied on a characteristic or an association end and return respectively the minimum multiplicity, the maximum multiplicity and the data type of a given characteristic or association end. These additional queries allow expressing the dependencies at the model layer. Consider the example outlined in Figure 48. The context of the invariants specifying dependencies at the model layer is `ModelLayer`. In the example, three model layer dependencies are expressed. Note that the model layer queries are applied on model elements, e.g. the characteristic `cardExpiryDate`, and not on objects of the user-object layer.



context Account **inv:** cardExpiryDate() = bankCard().expiryDate()

context ModelLayer **inv:**
Account.cardExpiryDate.dataType() = BankCard.expiryDate.dataType()

context ModelLayer **inv:**
Account.cardExpiryDate.minMultiplicity() =
Account.bankCard.minMultiplicity() * BankCard.expiryDate.minMultiplicity()

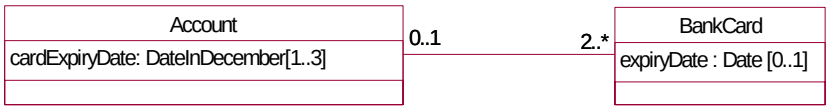
context ModelLayer **inv:**
Account.cardExpiryDate.maxMultiplicity() =
Account.bankCard.maxMultiplicity() * bankCard.expiryDate.maxMultiplicity()

Figure 48 Specifying model-layer dependencies

We are not fully convinced that these specifications are an elegant and a concise way to specify this kind of dependencies. They are only described to illustrate the idea of modelling dependencies on the model layer and as a step in promoting the adaptability of conceptual models.

12.3.2 Dependencies on the Model Layer differ from Dependencies on the Object Layer

One could argue that the last three invariants in Figure 48 are consequences of the first invariant and that it is not necessary to specify the last three invariants. This is not the case. Consider the example in Figure 49. Assume that a bankcard can have an expiry date and can be attached to maximum one account. The expiry date of a bankcard can be but must not be in December. An account must be linked with at least two bankcards. At least one and at most three of the bankcards linked with an account have an expiry date. In addition, the cardExpiryDate of an account must be in December. The dependency between the characteristics `cardExpiryDate` and `expiryDate` is specified in the invariant. Consequently, the dependencies between the multiplicities and data types of characteristics are not necessarily consequences of dependencies at the user object layer.



context Account **inv:** cardExpiryDate() = bankCard().expiryDate()

Figure 49 No dependencies between the multiplicities and the data types

13 Encapsulation during Analysis

13.1 Problem Description

The basic idea behind ***encapsulation*** or information hiding is to conceal irrelevant details for the potential users of a given element. The first advantage is that this promotes a better and natural way of communication with the users of the given element. A second advantage is that the hidden details can be modified without consequences for the way the given element is used. Encapsulation, still a key construct in object-orientation, is not discussed as such in the UML [OMG01] [Rumb99]. Only the notion of the

visibility of a model element can be specified in terms of annotations such as private or public. The principle of encapsulation is not only applied in software engineering, but also in almost all other disciplines. For instance, the manual of a washing machine describes how to turn it on but doesn't describe the internal working of the machine.

13.1.1 Encapsulation during Implementation

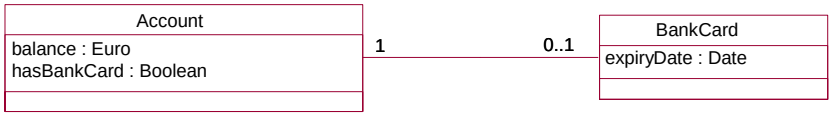
Encapsulation is adopted by all object-oriented programming languages [Stee99, p.83]. In object-oriented programming two kinds of encapsulation can be distinguished: hiding the data representation of an object and hiding the implementation of a method. The first kind of encapsulation is also called data hiding or data abstraction; the second kind is also called procedural abstraction. In addition to a better communication with the users of the given class, the advantages of encapsulation should be clear: the data representation and the implementation of the given class can be modified without the necessity for the users of the class to change their code. This promotes the adaptability of the software.

13.1.2 Encapsulation during Analysis

In object-orientation, the construct of encapsulation or visibility is considered to be meaningful during design and implementation. It is considered to be an irrelevant issue during analysis. According to the UML Reference Manual [Rumb99, p.432] visibility specifications for model elements appear during the design stage. During analysis, the model elements are assumed to be freely accessible. At first sight, this could seem logical because at the level of object-oriented analysis the data representation of an object and the implementation of a method are not and should not be worked out. Consequently, one could say that there is nothing to hide. At second sight, one could wonder whether some real-world facts are not a detail for certain users. Because encapsulation promotes adaptability, one could also pose the question if some real-world facts are not likely to change. These model elements could then be encapsulated.

13.2 Anti-Patterns

Consider an example outlined in Figure 50. All accounts have a balance and can be linked with at most one bankcard. All bankcards have an expiry date and are linked with exactly one account. The conceptual model seems to be a very good representation of the real-world facts.



context Account **inv:** hasBankCard() = (bankCard()→size() = 1)

Figure 50 No encapsulation

At second sight, one could pose the question if the multiplicity of the association end bankCard is not a detail for the user? In this example, the invariant (or the analyst who specifies the invariant) is considered to be a user. The invariant uses the fact that the maximum multiplicity is one. In fact, the multiplicity is a detail for the invariant. The multiplicity is not needed for the specification of the invariant. If the maximum multiplicity of the association end bankCard changes, then all specifications that "used" this multiplicity must be also changed, e.g. the specification of the invariant must be changed as well.

13.3 Patterns

The use of encapsulation at the level of analysis can promote adaptability. Consider the example outlined in Figure 51. The multiplicity of the association end bankcard is encapsulated. The encapsulation of a model element is denoted with the symbol E. This prevents the export of the multiplicity to the outside. Consequently, the multiplicity will not be used and can therefore be easily changed without the necessity to change other specifications. In the example, the specification of the invariant does not make use of the multiplicity of the association end bankCard. Consequently,

if the maximum multiplicity of the association end bankcard changes, the invariant must not be modified.

context Account **inv:** hasBankCard() = bankCard()→size() >= 1

Figure 51 The construct of encapsulation during analysis

Chapter 3

Patterns for Modelling Dynamic Properties

Whereas static properties deal with links between instances at any given moment, dynamic properties deal with changes of these links or with the migration of objects in sets. A dynamic property is a real-world fact that changes the state of the real world. Examples of dynamic properties are the deposit of a certain amount on an account and the closing of an account.

We consider dynamic properties as another important type of real-world facts to represent. In contrast with the functional approach [Somm92, p.219] and the data approach [Elma00, p.41], it is the merit of object-orientation that static as well as dynamic aspects can be equally emphasized and that it allows representing both kinds of properties in a single model.

The presence of restrictions in the real world, as for instance dependencies between static properties, makes the task of investigating how to specify behaviour only more complex. To specify valid behaviour, these restrictions should be taken into account. Moreover, during analysis, adequate specification of behaviour should not describe how to control these restrictions. As stated in [Roll92, p.4], modelling languages are often not rich enough to incorporate the dynamic aspects and the restrictions, or they are not well-defined and difficult to use. The specification of behaviour is strongly interconnected with the specification of restrictions and will therefore be discussed in a separate chapter about modelling restrictions.

The four sections in this chapter provide guidelines how dynamic properties can be modelled in an appropriate way. In section 1, dynamic properties and the way to specify and classify them is introduced. In section 2, the frame problem, which is the problem of specifying the unchanged static properties, is discussed in view of object-oriented analysis. In section 3, the solution of the frame problem is further refined for specialized events and composed

events. In section 4, the frame problem in case of the presence of dependencies between static properties is discussed.

1. Modelling Prototypes of Dynamic Properties

1.1 Problem Description

A dynamic property can be defined as a change of a link between instances, or as the migration of an object. Consider the examples outlined in Figure 1. A withdrawal is a change of a link between a savings account and a data value. The closing of a savings account is considered to be a migration of an object. For example, the closed savings account can be migrated to the set of closed savings accounts. The set of closed savings accounts is considered to be irrelevant in this example. The different kinds of dynamic properties are further discussed in this section.

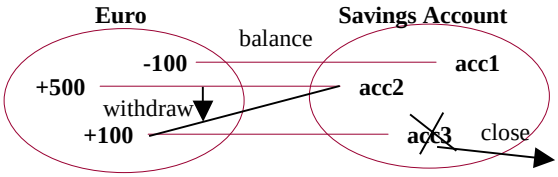


Figure 1 Dynamic properties

1.2 Patterns

1.2.1 Events

A *prototype* of dynamic properties of the same kind is modelled by an ***event***. An event describes how the state of the model may evolve over time. An event occurs at a point in time and has no duration [Rumb99, p.68].

An event does not model a set of all its occurrences. As an example, a withdraw-event does not model a set of all accomplished withdrawals; an open-event does not model a set of all openings of accounts. A withdraw-

event represents a prototype of a withdrawal, describing what it means to withdraw money from some account. In the same way, an open-event models a prototype of an opening of an account. Classes are used to model sets of objects. Further investigation is needed about the distinction between classes and events and about the use of classes versus events.

In the UML, events are used in state diagrams. Our construct of an event does not (fully) match with the UML construct of an event [Brow02, p.347]. Our construct of an event matches better with the UML construct of an operation, as far as it is not a query [OMG01, 2-72]. Consequently, our events can be used in class diagrams.

1.2.2 Specification of Events

As for characteristics, events will be attached to the involved classes instead of to the involved data types. In other words, events are structured as characteristics. Consider the example outlined in Figure 2. The event `withdraw()` will be attached to the class `Account` and not to the data type `Euro` nor to both classifiers. Consequently, one could say that objects have not only static properties such as a balance, but also dynamic properties, such as a withdrawal. Events are modelled in the second compartment of the class while characteristics are modelled in the first.

context Account :: withdraw(amount: Euro)
post: balance() = balance()@pre – amount -- *the effect of an event*

Figure 2 The specification of an event

The ***effect of an event*** specifies how the new state of the model can be derived from the old state of the model. The effect of an event is specified by means of a postcondition. Reconsider the example in Figure 2. If a withdrawal occurs, the balance of the account is decreased with the given

amount. Events can have explicit arguments. The way to specify the effect of events is further discussed in this chapter and in the chapter about modelling restrictions.

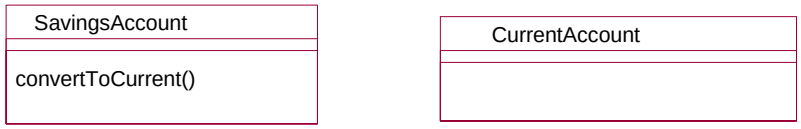
It is well agreed that at the level of analysis, dynamic properties are best described in a ***declarative way*** [Vane01, p.462]. At the level of analysis, an event must only specify its effect without wondering how this effect can be accomplished [Steeg00, p.151]. Several assertions in a postcondition are connected with an implicit conjunction (logical and) [Meye97, p.337]. Consequently, the order of the assertions is arbitrary. Notice, that an implicit conjunction is not foreseen in the OCL [OMG01].

Events are always applied to objects. The expression `self.event()` denotes the application of the event `event()` to the object `self`. Such expression can occur in the specification of the effect of other events [section 1.2.4] and in the specification of functional requirements [chapter 7]. Events can be only applied on objects of the class to which the event is attached.

1.2.3 Mutators and Migrants

A first classification of events can be made in terms of mutators and migrants. A ***mutator*** models a change of a link between objects or between objects and data values. The event `withdraw()` in the Figure 2 is an example of a mutator. In the example, the mutator `withdraw()` changes the link between the account to which it is applied and a data value of type Euro.

A ***migrant*** models the migration of an object from a certain set into another set. Consider the example outlined in Figure 3. Assume that a savings account can be converted into a current account. The event `convertToCurrent()` applicable to a savings account models such a conversion. Notice that this event is attached to the class `SavingsAccount` although two classes are involved. How events (and characteristics) are modelled when more than one class is involved is discussed in chapter 4.



context SavingsAccount :: convertToCurrent()
post: not oclIsTypeOf(SavingsAccount)
oclIsTypeOf(CurrentAccount)

context SavingsAccount **inv:** not oclIsTypeOf(CurrentAccount)@pre
-- ***context*** *CurrentAccount* ***inv:*** *not oclIsTypeOf(SavingsAccount)@post*

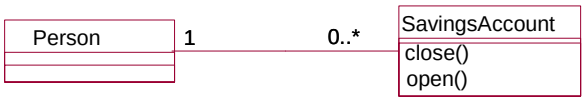
Figure 3 A migrant

No assumption is made about dynamic or static classification when two or more classes are represented in a model. A class models only a set of objects. This promotes the principle of abstraction. We fully agree with [Rumb99, p.54], where it is stated that there is no logical necessity to forbid dynamic classification. The restriction is primarily intended to make the implementation of object-oriented programming languages easier but from our point of view dynamic classification is a valuable modelling possibility. In EROOS [Steeg00], dynamic classification is not allowed.

The invariant in the example of Figure 3 specifies that current accounts cannot be converted into savings accounts. Notice that, in contrast with the OCL [OMG01, p.6-22], we allow the use of the `@pre` postfix in the specification of invariants. In the OCL, this annotation is only allowed in a postcondition. The use of the `@pre` postfix allows modelling a more extensive and richer set of restrictions. For instance, a restriction that the balance of an account can only increase is now modelled as `inv: balance() >= balance()@pre`. It is assumed that invariants, where the `@pre` postfix is used, are satisfied by definition for constructors. The introduction of the postfix `@post` allows attaching the invariant at the class `CurrentAccount` and is illustrated in the comment of Figure 3.

From our point of view, destructors and constructors are special kinds of migrants. A ***destructor*** represents the removal from an object in a set and makes abstraction of the class to which the given object migrates. The class

to which the given object migrates is considered to be irrelevant and is therefore not represented in the conceptual model. Consider the example outlined in Figure 4. The event `close()` models the closing of a savings account and the removal of this object from the set of savings accounts. The account migrated from the set of savings accounts to another not represented set of objects.



context SavingsAccount :: close()
post: not oclIsTypeOf(SavingsAccount)

context SavingsAccount :: open()
post: oclIsTypeOf(SavingsAccount)

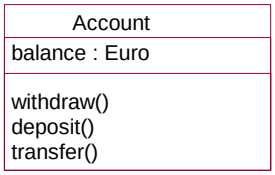
Figure 4 Constructors and destructors

A ***constructor*** represents the insertion of an object in a set and makes abstraction of the class from which the given object is migrated. The class from which the given object is migrated is considered to be irrelevant and is therefore not represented in the conceptual model. Reconsider the example outlined in Figure 4. The event `open()` models the opening of a savings account and the insertion of an object in the set of savings accounts. The account migrated to the set of savings accounts. The class from which the savings account is migrated is abstracted. Notice that constructors are applied on objects that are not represented in the conceptual model. Further investigation is needed about the application of constructors.

1.2.4 Compound Events and Non-Compound Events

Another classification can be made in terms of compound events and non-compound events. A ***compound event*** has at least one other event in its postcondition. A ***non-compound event*** has no events in its postcondition. The effect of a non-compound event is specified in terms of queries. Consider the example outlined in Figure 5. The events `withdraw()` and

`deposit()` are examples of non-compound events, while the event `transfer()` is an example of a compound event.



context Account :: withdraw(amount : Euro)
post: balance() = balance()@pre - amount

context Account :: deposit(amount: Euro)
post: balance() = balance()@pre + amount

context Account :: transfer(account: Account, amount: Euro)
post: withdraw(amount) -- *balance() = balance()@pre - amount*
account.deposit(amount) --*account. balance() =account. balance()@pre - amount*

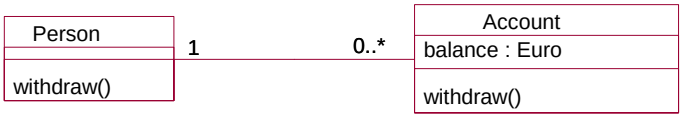
Figure 5 Compound and non-compound events

The OCL forbids compound events [OMG01, p.6-2 and p.6-11]. It is argued that the OCL is not a programming language; therefore, it is not possible to write program logic or flow control. It is impossible to invoke processes or to activate non-query operations. From our point of view, this is not a valid reason to forbid the reuse of events in order to specify compound events. Obviously, the OCL is a specification language and not a programming language but the postcondition of a compound event must be read as an assertion and does not indicate any flow control or program logic. Moreover, as illustrated in the comment of the event `transfer()`, a compound event can always be reformulated in terms of queries.

The reuse of an event allows the analyst to succinctly specify a compound event, and allows specifying the semantics of an event in terms of a series of other events. Compound events promote reusability and adaptability. Reusability increases, because events can be reused and the specification of existing postconditions must not be repeated. Adaptability increases, because

if the effect of an event changes, e.g. `withdraw()`, only one postcondition must be changed.

An event applied to a collection of objects implies that this event is applied on each element of this collection. An event applied to an empty collection does not change the state of the model. Consider the example outlined in Figure 6. The event `withdraw()` in the class `Person` decreases the balance of all accounts of the person to which the event applies.

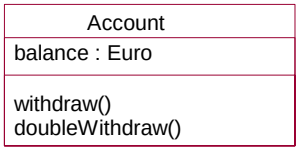


context Account::withdraw(amount)
post: balance() = balance()@pre – amount

context Person :: withdraw(amount)
post: account().withdraw(amount)
--account()→forAll(account|account.withdraw())

Figure 6 An event applied on a collection

Due to the declarative reading of the specification, some problems can occur if events are reused to specify a compound event. Consider the event `doubleWithdraw()` in the example outlined in Figure 7. Although the semantics of this event is that two withdrawals occur, this cannot be modelled as such. If the event `withdraw()` is reused several times in the event `doubleWithdraw()`, then the assertions in this last event contradict each other. A possible solution is indicated in the specification language Z [Dill90, p.67]. This specification language introduces therefore the semicolon operator, offering the ability to introduce intermediate states in specifications of events.



context Account :: withdraw()
post: balance() = balance()@pre – amount

context Account :: doubleWithdraw(amount1: Euro, amount2: Euro)
post: balance() = balance()@pre – amount1 – amount2
-- *withdraw(amount1)*
-- *withdraw(amount2)*

Figure 7 Problems with the reuse of events

1.2.5 Conclusion

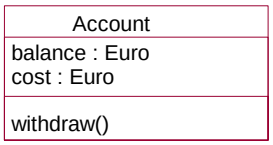
Prototypes of dynamic properties are modelled by events. The effect of an event is described in a declarative way with the help of a postcondition. As characteristics, events are structured around the involved classes. Mutators, and migrants are different kind of events. Constructors and destructors are considered to be abstracted migrants. The presence of migrants assumes that dynamic classification is supported. Furthermore, and in contrast with the OCL, it is possible to define compound events in terms of other compound and non-compound events.

2. The Frame Problem during Analysis

2.1 Problem Description

A dynamic property is a real-world fact and a change in the state of the real world, but a dynamic property leaves also most real-world facts unchanged. How can these unchanged real-world facts be modelled at the level of analysis? Consider the example outlined in Figure 9. A withdrawal decreases the balance of an account but does not change the individual cost of an account, nor does it open a new account or close an existing account. In

other words, a withdrawal decreases the balance of the account to which it is applied and nothing else in the state of the model is changed. How can this "nothing else changed" be specified?



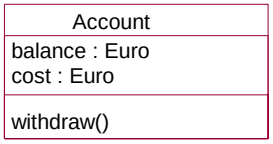
context Account :: withdraw(amount : Euro)
post: balance() = balance()@pre – amount

Figure 8 The frame problem

In knowledge engineering, extra assertions stating that nothing else changes are called the ***frame axioms*** [Brac01, p.260]. The problem of stating succinctly (and reasoning with) a large number of frame axioms is called the ***frame problem***. Obviously, the analyst must be able to state these extra assertions in one way or another in order to represent the real world.

2.2 Anti-Patterns

A possible approach is to specify explicitly the frame axioms. Consider the specification of the event `withdraw()` in Figure 9. Not only the effect of the event is explicitly specified, but also the frame axioms for the event.



context Account :: withdraw(amount : Euro)
post: -- *effect*
balance() = balance()@pre – amount
-- *frame axioms*
Account.AllInstances→ exluding(self)→forAll(balance() = balance()@pre)

```
Account.AllInstances→ forAll(cost() = cost()@pre)
Account.AllInstances = Account.AllInstances@pre
```

Figure 9 Explicitly specifying the frame axioms

Requiring the analyst to specify the frame axioms explicitly makes analysis a lengthier and more error-prone process [Borg95, p.789]. Furthermore, explicitly specifying the frame axioms do not support extendibility because adding a new characteristic, association or class leads to specifying an additional frame axiom in each event.

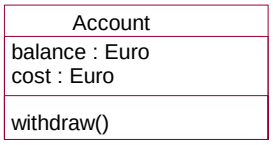
2.3 Patterns

2.3.1 The Inertia Principle

The ***principle of inertia*** reflects the tendency of matter to remain at rest except if some force is acted upon this matter. In analysis, we reformulate this principle as: all static properties remain unmodified unless otherwise specified. During analysis, the frame problem is resolved by adopting the inertia principle. Consequently, the postconditions of the events will only specify possible differences between the new state and the old state of the involved objects. All aspects of the old state that are left untouched are not explicitly enumerated. The assertions are implicitly added to the specification of the event.

The adoption of the principle of inertia is illustrated in the example of Figure 10. A withdrawal decreases the balance of account with a given amount. This is explicitly specified in the postcondition. In other words, the effect of an event is explicitly specified in the postcondition. A withdrawal does not change the cost of an account. This is implicitly specified by the adoption of the inertia principle. In other words, the frame axioms are implicitly specified.

In the explicated specification of the event, the implicit assertions are made explicit. An explicated postcondition (epost) is used to represent the implicit and explicit assertions.



context Account **inv:** cost() <= 30 Euro

context Account :: withdraw(amount : Euro)
post: balance() = balance()@pre – amount

Explicated specification of the event:
context Account :: withdraw(amount : Euro)
epost: -- *effect*
self.balance() = self.balance()@pre – amount and
-- *frame axioms*
Account.AllInstances→ exluding(self)→forAll(balance() = balance()@pre) and
Account.AllInstances→ forAll(cost() = cost()@pre) and
Account.AllInstances=Account.AllInstances@pre

Figure 10 The inertia principle and the explicated specification of an event

Notice that the adoption of the inertia principle does not support the principle of abstraction. An analyst is not capable to specify an event with relevant and irrelevant effects. For example, a relevant effect of the event `withdraw()` is the decrease of the balance. Other possible effects of this event, e.g. an increase of the cost, are irrelevant. The adoption of the inertia principle excludes the possibility of irrelevant effects of an event. As stated in [Borg95, p.804], in some circumstances, the analyst may want the freedom not to use the frame axioms. We are not convinced that there exist such circumstances during the analysis phase of the software development life cycle.

2.3.2 The Frame Problem during Design and Implementation

A difficulty with the adoption of the inertia principle is finding an effective and general algorithm for computing the frame axioms [Borg92, p.16]. During design and implementation, the formal specifications are not only used to precisely specify behaviour of a procedure but also to prove

properties about specifications. Typical kinds of proofs are proofs that an implementation of a procedure meets its specification, and proofs that a procedure maintains the invariants. For both kinds of proofs the frame axioms are needed (and must be computed or explicitly specified).

Reconsider the example outlined in Figure 10. If the frame axioms are absent, an implementation of the procedure `withdraw()` can change the cost of an account without violating its postcondition. Furthermore, it cannot be proved that this procedure maintains the invariant `cost() <= 30 Euro` if there is no frame axiom that guarantees that this procedure does not change the cost of an account.

Obviously, the first kind of proof is not at stake during analysis. Also the second kind of proof will not be at stake during analysis. As it will be discussed in the chapter about restrictions, an event will be rejected in case it would violate a business rule. In case of rejection of an event, the state of the model remains unchanged. Consequently, in analysis, the problem of finding an effective and general algorithm to compute the frame axioms is of minor importance.

More generally, during object-oriented analysis the focus lays not on proving properties but on representing real world facts. In analysis, the adequacy of formal specification language is much more determined by its notational suitability than by its capacity to support formal treatment. Notational suitability is defined as the degree to which a specification language makes it possible for the analyst to express assertions in a precise yet simple, concise, understandable, adaptable and extendible way. The capacity to support formal treatment is defined as the extent to which a specification language provides a foundation for formally proving properties [Borg95, p.786]. Notice, that in contrast with analysis, knowledge engineering focuses on reasoning and proving. In knowledge engineering, this focus can lead to performance problems due to extensive computation (for instance, with frame axioms [Miss98, p.28]). In analysis, computation is not the first concern.

3. Refinement of the Inertia Principle

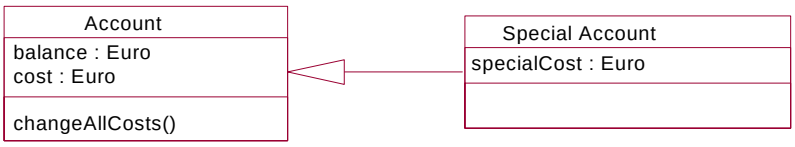
3.1 Problem Description

During object-oriented analysis, events can be specialized. Problems could arise if the inertia principle is strictly applied at the level of generalization. In this case, no additional modifications can be specified at the level of specialization. An analogous problem occurs in case events are compound.

3.2 Patterns

3.2.1 Specialization of Events

Consider the example outlined in Figure 11. The event `changeAllCosts()` changes the cost of an account at the level of generalization. If the inertia principle is strictly applied, this event specifies further that nothing else change. This cannot be contradicted at the level of specialization. Consequently, at the level of specialization this event cannot modify other static properties. In EROOS [Stee00, p.149], the inertia principle is strictly applied at the level of generalization: strengthening the specification of an event may not be interpreted as an additional effect, but is defined as a more accurate description of the effect. Consequently, an event can only be specialized if some non-determinism is involved in the specification of the event at the level of generalization.



context Account :: changeAllCosts(percentage: Real)
post: cost() = (1+percentage) * cost()@pre

context SpecialAccount :: changeAllCosts(percentage: Real)
post: specialCost() = (1+percentage) * specialCost()@pre

Explicated specification of the event:

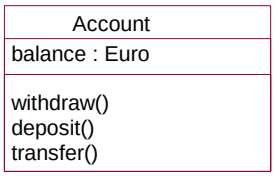
context SpecialAccount :: changeAllCosts(percentage: Real)
epost: -- *effect*
self.cost() = (1+percentage) * self.cost()@pre and
self.specialCost() = (1+percentage) * self.specialCost()@pre and
-- *frame axioms*
Account.AllInstances→ exluding(self)→forAll(cost() = cost()@pre) and
SpecialAccount.AllInstances→exluding(self)→forAll(
specialCost() = specialCost()@pre) and
Account.AllInstances→ forAll(balance() = balance()@pre) and
Account.AllInstances=Account.AllInstances@pre

Figure 11 Specialized events and the inertia principle

We are not in favour of the strict application of the inertia principle at the level of generalization, which prevents additional modifications at the level of specialization. Therefore, the inertia principle is only applied *after* inheritance. Consequently, the frame axioms are not inherited as such but are added to the explicit assertions after inheritance. Obviously, the effect of an event is inherited. Reconsider the example in Figure 11. The event `changeAllCosts()` at the level of specialization inherits the effect of this event specified at the level of generalization. After inheritance, the principle of inertia is applied. Notice that the event `changeAllCosts()` still guarantees an unchanged balance for each account.

3.2.2 Compound Events

Consider the example outlined in Figure 12. The event `withdraw()` only modifies the balance of the account to which it is applied. The conjunction of the frame axioms of this event with the other assertions of the event `transfer()` leads to contradicting assertions in the specification of the event `transfer()`. A same solution as for inheritance is applied; namely, the application of the inertia principle *after* the composition of events. In the same way, frame axioms are not inherited and are also not conjoined in case of compound events.



context Account :: withdraw(amount : Euro)
post: balance() = balance()@pre - amount

context Account :: deposit(amount: Euro)
post: balance() = balance()@pre + amount

context Account :: transfer(account: Account, amount: Euro)
post: withdraw(amount)
account.deposit(amount)

Explicated specification
context Account :: transfer(account: Account, amount: Euro)
epost: -- *effect*
self.balance() = self.balance()@pre - amount and
account.balance() = account.balance()@pre + amount and
-- *frame axioms*
Account.AllInstances→ exludingAll({self,account})→forAll(balance() = balance()@pre) and Account.AllInstances=Account.AllInstances@pre

Figure 12 Compound events and the inertia principle

3.2.3 Conclusion

The application of the principle of inertia is refined: the principle is applied after inheritance and after the composition of events in order to allow additional modifications at the level of specialization and in order to avoid inconsistency in composed events.

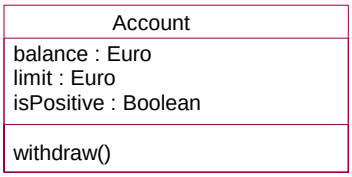
4. Relaxation of the Inertia Principle

4.1 Problem Description

The adoption of the inertia principle during analysis, which expresses that static properties remain untouched unless otherwise specified, poses problems in case of dependencies between static properties. When the inertia principle is strictly adopted, an event that changes a static property must sometimes change other static properties in the dependency as well.

4.2 Anti-Patterns

A possible approach is to specify explicitly the possible change of the derived static property. Consider the example outlined in Figure 13. An account has a balance, a limit and can be positive or not. Two dependencies between these static properties exist; they are represented by invariants. Consider the specification of the event `withdraw()` in this model. The intention of this event is to decrease the balance of an account. Aside the specification of the decrease of the balance, the possible change of the static property "isPositive" is also specified.



context Account **inv**: balance() >= limit()
context Account **inv**: isPositive() = (balance() >= 0 Euro)

context Account :: withdraw(amount : Euro)
post: balance() = balance()@pre - amount
isPositive() = (balance() >= 0 Euro)

Explicated specification of the event:
epost: -- *effect*
self.balance() = self.balance()@pre – amount and
self.isPositive() = (self.balance() >= 0 Euro) and

```
-- frame axioms
Account.AllInstances→ exluding(self)→forAll(balance() = balance()@pre) and
Account.AllInstances→ exluding(self)→forAll(isPositive() = isPositive()@pre)
and Account.AllInstances→ forAll(limit() = limit()@pre) and
Account.AllInstances=Account.AllInstances@pre
```

Figure 13 No relaxation of the inertia principle

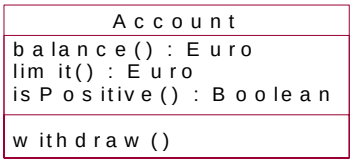
If the inertia principle is not relaxed, one must also specify a possible modification of the static property "isPositive". Generally, this means a lot of extra work for the analyst and the risk to forget some modifications. However, if such modifications are not taken into account, then the specification of the event can contradict the specification of the invariant.

Extendibility is also not well supported in such cases. When a characteristic `isNegative` is added to the model, the existing specification of the event `withdraw()` must be adapted. Furthermore, the analyst has the impression that he is obliged to specify two times the same fact: the assertion concerning changes to the property "isPositive" in the event `withdraw()` is already represented in the invariant for that property.

4.3 Patterns

4.3.1 Relaxed Inertia Principle

One could argue that it is not needed to specify the extra modification because invariants must always be satisfied. Consequently, invariants must be satisfied after the occurrence of an event and they can be implicitly added to the postcondition. This is correct and the implicit addition of the invariants can help to avoid extra specifications. Consider the approach in Figure 13 where the invariants are implicitly added to the postconditions. It is no longer necessary to specify the possible change of the static property "isPositive"; however, other problems with the adoption of the inertia principle occur. These problems will be discussed in the next paragraphs.



context Account **inv**: balance() >= limit()
context Account **inv**: isPositive() = (balance() >= 0 Euro)

context Account :: withdraw(amount : Euro)
post: balance() = balance()@pre - amount

Explicated specification of the event:
epost: -- *effect*
self.balance() = self.balance()@pre – amount and
-- *invariants*
Account.AllInstances→ forAll(balance() >= limit()) and
Account.AllInstances→ forAll(isPositive() = balance() >= 0 Euro) and
-- *frame axioms*
Account.AllInstances→ exluding(self)→forAll(balance() = balance()@pre) and
Account.AllInstances→ exluding(self)→forAll(isPositive() = isPositive()@pre)
and Account.AllInstances→ forAll(limit() = limit()@pre) and
Account.AllInstances=Account.AllInstances@pre

Figure 14 The frame axiom and dependencies

What is the consequence of implicitly adding the invariants to the postcondition for the inertia principle? The formulation of the inertia principle becomes ambiguous. What is the meaning of "unless otherwise specified"? Does this mean "unless explicitly otherwise specified", without taking into account the implicitly added invariants, or "unless implicitly and explicitly otherwise specified". In the latter case, the implicitly added invariants are taken into account. Are other interpretations more adequate? Below, four different interpretations are discussed:

- If the inertia principle is strictly interpreted, meaning that all static properties are left untouched unless explicitly specified, the implicitly added invariants and the frame axioms can contradict each other. For instance, after a withdrawal the invariant `isPositive() = (balance() >= 0 Euro)` can contradict the frame axiom

`isPositive() = isPositive()@pre.` This interpretation is too strict.

- If the adoption of the inertia principle means that all static properties are left untouched unless implicitly or explicitly specified otherwise, it is not guaranteed that the balance or the limit of an arbitrary account is left untouched. For example, the balance or limit of an arbitrary account can be changed. This interpretation is too relaxed.

- The inertia principle could assume that when a static property, which is (directly or indirectly) present in a dependency, is explicitly modified, other static properties of these dependencies can change as well. With this assumption, it is not guaranteed that the limit of the account, which changes its balance, is left untouched. Consequently, the disadvantage of this interpretation is that the unchanged limit of the account must be specified explicitly. The interpretation is still too relaxed.

- The adequate relaxation of the inertia principle assumes that if a static property, which is (directly or indirectly) present in a dependency, is explicitly modified, then other *derived* static properties in these dependencies can also change. A derived static property is defined as a static property that can be computed in terms of the other static properties in the dependency. Reconsider the frame axioms in the second approach of Figure 13. Every static property is left untouched except the balance of the involved account because this modification is explicitly modelled, and except for the static property “isPositive” of the involved account because this property can be computed in terms of the balance of the involved account. The limit of the involved account cannot be changed because the limit of an account cannot be computed in terms of the balance of that account.

Notice that the effect of the postcondition `balance = balance()@pre – amount` can contradict the invariant `balance() >= limit()`. This problem is discussed in the chapter about restrictions.

A model element is derived if its query is present in a dependency and if its query can be computed for all possible objects in terms of all the other queries that are present in the dependency. A static property is derived if its model element is derived. Consider the example in Figure 15. The characteristic `cardExpiryDate` is derived because its query can be computed in terms of the other queries `bankcard()` and `expiryDate()`. In other words, if one knows the bankcard of an account and the expiry date of this bankcard, the card expiry date of the account can be calculated. The association between the class `Account` and the class `BankCard` is not derived. The query `bankcard()` cannot be calculated in terms of the queries `expiryDate()` and `cardExpiryDate()`. In other words, if one knows the expiry date of the bankcards and the card expiry date of accounts, the links between accounts and bankcards cannot be computed. Analogously, the characteristic `expiryDate` is derived because its query can be computed in terms of the query `cardExpiryDate()` and `bankcard()`.

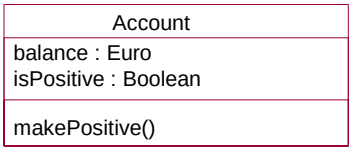
context Account **inv:** cardExpiryDate() = bankCard().expiryDate()

Figure 15 Derived model elements

4.3.2 Consequences

A first consequence is that, normally, derived static properties cannot be used to model non-deterministic events. Consider the example outlined in Figure 16. The event `makePositive()` is specified in terms of the balance of an account. If this event would be specified in terms of the query `isPositive()`, then the balance of the account would not be modified and the invariant would be contradicted. An advantage of this consequence is that the non-deterministic aspects in the specification of an event are more perceptible. Without the invariant, it is not clear that an event with a postcondition `isPositive() = True` is non-deterministic. A disadvantage

is that the specification of the postcondition cannot be made more succinct with the help of a derived static property.



context Account **inv:** isPositive() = (balance() >= 0 Euro)

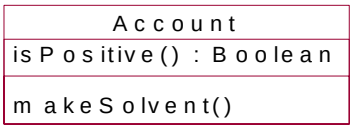
context Account :: makePositive()
post: balance() >= 0 Euro -- *instead of isPositive() = True or simply isPositive()*

Figure 16 Derived model elements and non-derministic events

If a conceptual model is extended, a static property can become a derived one. As a second consequence, such a static property that becomes derived can no longer be used in the specification of an event. The initial specification of the event would violate an invariant, as explained in the previous paragraph. From our point of view, this is not a disadvantage. If a model is further detailed, it is quite logic that some events must be further detailed. In our approach, the analyst is forced to reconsider and to refine the semantics of the event.

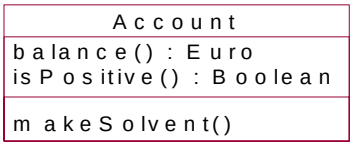
Consider a small example in Figure 17. Assume an initial model with a characteristic `isPositive` and an event `makeSolvent()`. To make an account solvent, it must be made positive. Assume further that later on, the class `Account` is extended with a characteristic `balance` and the well-known dependency `isPositive() = (balance() >= 0 Euro)`. The specification of the event `makeSolvent()` is no longer valid. The analyst is now forced to further investigate this event. At that time, the analyst may find out that in order to make an account solvent, its balance must be set to 25 Euro.

Initial Model



context Account :: makeSolvent()
post: isPositive()

Extended Model



context Account **inv**: isPositive() = (balance() >= 0 Euro)

context Account :: makeSolvent()
post: balance() = 25 Euro -- *isPositive()*

Figure 17 Derived static properties and extendibility

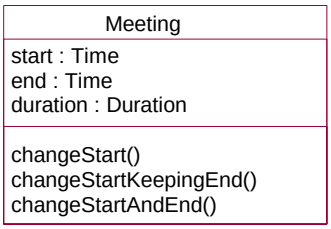
4.3.3 Dependencies with several derived Static Properties

The relaxation of the inertia principle states that all static properties remain untouched, except for static properties that are explicitly changed and except for derived static properties that are involved in a dependency in which the static property, which is explicitly changed, is also involved. Does this reformulation of the inertia principle poses problems in case events must be specified in which a static property is changed from which several derived static properties depend?

If several derived static properties are involved in a dependency, then all of them may change if one of the static properties involved in this dependency is explicitly modified. This can lead to non-deterministic events or to an explicit specification that such a derived static property is left untouched.

Consider the example outlined in Figure 18. The event `changeStart()` is non-deterministic because the end and the duration of a meeting are both

derived static properties and the inertia principle does not apply to these properties. In the postcondition of the event `changeStartKeepingEnd()` it is explicitly specified that the end of the meeting is left untouched. The duration of the meeting is then implicitly adapted in such a way that the invariant still holds immediately after the event has occurred. Notice that for the event `changeStartEnd()` it is not necessary to specify the change of the duration of the meeting.



context Meeting **inv:** end() = start() + duration()

context Meeting::changeStart(start: Time) -- *non-deterministic event*
post: start() = start

context Meeting::changeStartKeepingEnd(start: Time) -- *deterministic event*
post: start() = start
end() = end()@pre

context Meeting::changeStartAndEnd(start: Time, end: Time) -- *deterministic event*
post: start() = start
end() = end

Figure 18 More than one derived static property in a dependency

4.3.4 Conclusion

The inertia principle helps the analyst to specify events in a succinct and easy way. The relaxation of the inertia principle and the implicit addition of the invariants to the postcondition of an event keep the invariants satisfied without the obligation of the analyst to explicitly specify changes to derived characteristics. The inertia principle is only applied after specialization and

after composition of events. This makes it possible to specify compound events and to specialize events with additional effects.

Chapter 4

Patterns for Modelling Properties

Characteristics model sets of static properties while prototypes of dynamic properties are modelled by events. In this chapter, some patterns valid for both kinds of properties will be introduced. Characteristics as well as events are considered to be ***features***. Consequently, features are model elements and the unifying construct to model properties in general.

In section 1, the representation of properties where more than one object is involved is discussed. In section 2, the possible use of features in which no objects or only artificial objects are involved is presented.

1 Features involving more than one Object

1.1 Problem Description

Several objects and data values can be involved in a property. Consider the example in Figure 1. The static property “totalBalance” indicates a possible link between the total balance of a person at a certain bank. How can this kind of properties be modelled?

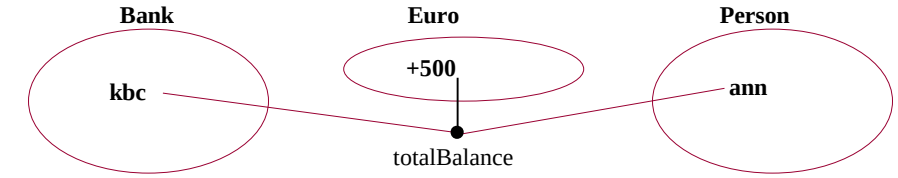


Figure 1 Properties with more than one object involved

In each of the following subsections, possible modelling alternatives are discussed. It has already been stated that features modelling properties in which objects are involved, are attached to classes instead of to data types. Notice that an association models a set of links between objects and that the association is attached to all involved classes.

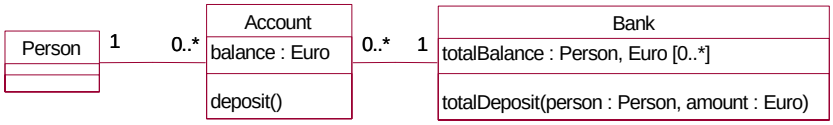
1.2 Anti-Patterns

1.2.1 Attaching the Features to only one involved Class

Attaching features to classes works well as long as one object is involved. Problems arise when more than one object is involved. When this is the case, the question rises to which class the feature will be attached. As an example consider Figure 2, in which the characteristic `totalBalance` is defined modelling the total balance of a certain person at a certain bank, and the event `totalDeposit()`, for depositing some amount of money on all the accounts owned by a certain person at a certain bank. In the first alternative, these features are attached to the class `Bank` while in the second alternative they are attached to the class `Person`.

Notice that the OCL was designed first of all for expressing constraints, and not that much for specifying queries. In [Akeh01], some extensions for the OCL are proposed in order to be useful as a query language and to reach the expressive power of relational algebra. Examples of added constructs are tuples, project operations and product operations. The next version of OCL, version 2.0, should take into account these extensions [Warm03]. The notion of a tuple is used in the specification of the invariants.

Alternative one

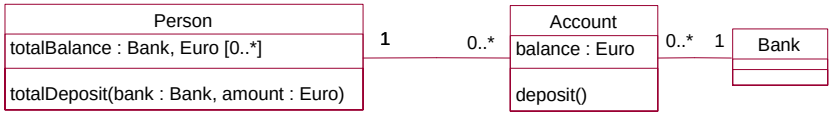


context Account :: deposit(amount: Euro)
post: balance() = balance()@pre + amount

context Bank :: totalDeposit(person: Person, amount: Euro)
post: account()→select(person() = person).deposit(amount)
-- *person.account()→select(bank() = self).deposit(amount)*

context Bank **inv:** totalBalance() =
account().person()→asSet→collect(p| Tuple {
p: Person,
p.account().intersection(self.account()).balance()→sum() : Euro})

Alternative two



context Account :: deposit(amount: Euro)
post: balance() = balance()@pre + amount

context Person :: totalDeposit(bank: Bank, amount: Euro)
post: account()→select(bank() = bank).deposit(amount)
-- *bank.account()→select(person() = self).deposit(amount)*

context Person **inv:** totalBalance() =
account().bank()→asSet→collect(b| Tuple {
b: Bank,
b.account().intersection(self.account()).balance()→sum() : Euro }

Figure 2 Features attached to one involved class

Which alternative an analyst should choose? Evaluating the specification of the invariants or the specification of the postcondition of the events in the modelled alternatives will not help. These specifications are each other's mirror image.

One could argue that if the user of the future system is a bank, then the feature could be attached to the class `Bank`. If the user were a person, the feature would then be attached to the class `Person`. Attaching features to the class that models the user of the system has at least two disadvantages:

- Firstly, we want to make abstraction of possible users of the system. In a conceptual model, real-world observations are modelled, not the users of the future system. It is not important if the user will be a bank, an

individual, the fiscal authority, … or a combination of such users. It is possible that both the bank and an individual person will use the system later on. By making abstraction of the users of the system, it is not necessary to change the conceptual model if the users of the system change.

- Furthermore, this rule would lead to an attachment of all features representing functional requirements, to the class "`User`". This would lead to an unbalanced distribution of properties in the conceptual model. Notice, that in the above conceptual model the user of the system is not necessarily represented.

Consider the specification of the event `totalDeposit()` in both alternatives. A choice about the class of the event suggests a certain design decision and may therefore not be taken during object-oriented analysis. Furthermore, different alternatives for the specification of the event exist even if a class is chosen. The alternatives are given in comment. The object where the specification starts can be the implicit object `self` or the object that is given as explicit argument.

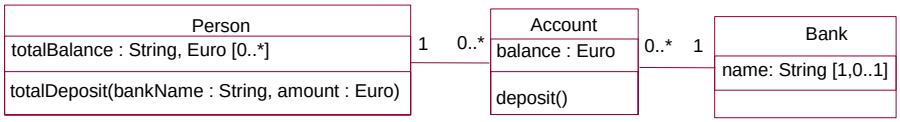
Consider the characteristic `totalBalance` in both alternatives. These characteristic models a set of links between persons, banks and data values of type Euro. On top of this representation of real-world facts, functional requirements can be built. As an example, given a certain bank and a certain person return the total balance of all the accounts of that person at that bank. As another example, given a certain bank and a certain amount of money, return the set of persons whose total balance at that bank is not below the given amount. In the early stages of conceptual modelling, one must not only make abstraction of the future users of the system, but also of the functional requirements. In a conceptual model, real-world facts must be modelled at the first place and not functional requirements for the ultimate system. The modelling of functional requirements is further discussed in a separate chapter. If the characteristic is attached to the class `Person`, an implicit query `totalBalance(): Bank,Euro [0.*]` is given that can be applied on a certain person. If the characteristic is attached to the class `Bank`, a query `totalBalance(): Person,Euro [0.*]` is given that can be applied on a certain bank. An analyst could tend to attach the characteristic to a class in

function of the demanded functional requirements. This tendency should be avoided.

In the example, two alternatives are possible modelling the same external world. This violates the principle of no-choice. In order to guide the analyst, we strive at only one model for the same reality. The decision, to which class a feature belongs involving several objects, adds no semantics to the conceptual model and would be an arbitrary one. As for associations, we do not want to distinguish between classes (or objects) involved in a feature.

1.2.2 Using unique Data Values

One could argue that the choice between classes should not be made if a unique data value is used instead of an object. Consider the example outlined in Figure 3 where the features `totalDeposit()` and `totalBalance` are attached to the class `Person` and where it is assumed that each bank has a unique name. The name of the banks serves to identify a bank.



context Account :: deposit(amount: Euro)
post: balance() = balance()@pre + amount

context Person :: totalDeposit(bankName: String, amount: Euro)
post: account()→select(bank().name() = bankName).deposit(amount)

context Person **inv:** totalBalance() =
account().bank()→asSet→collect(b| Tuple {
b.name(): String,
b.account().intersection(self.account()).balance()→sum() : Euro }

Figure 3 A feature with a unique data value

This approach has several disadvantages:

- Firstly, the principle of abstraction is violated: the analyst is forced to search for a unique data value for a bank.

- Secondly, the choice between classes must still be made. As an example, it is also possible to search for a unique data value to distinguish between persons and to attach the feature `totalDeposit(uniqueDataValue-ForPerson:DataType,amount:Euro)` to the class `Bank`.

- Another disadvantage is that the characteristic `totalBalance` does not represent ternary links between persons, banks and data values of type Euro but ternary links between persons and data values of type Euro and String.

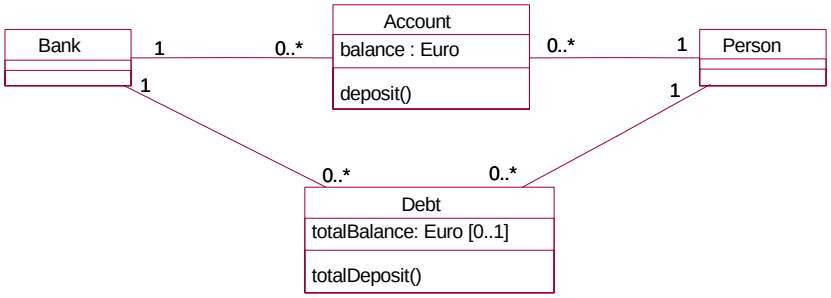
- Another objection is that in object-orientation one prefers to work with objects and not with unique data values.

1.2.3 Features attached to a non-involved Class

One could argue that it is always possible to find a class to which features such as `totalBalance` and `totalDeposit` can be attached so that an arbitrary choice between involved classes or the use of unique data values can be avoided. This is correct. Consider as examples the models in Figure 4, Figure 5 and Figure 6.

A class `Debt` is added to the conceptual model of Figure 4. The class represents the set of debts. A person can have a debt at a bank. This model violates the principle of abstraction. An analyst is forced to further investigate the real world in order to find a possible class to attach the features. Another important objection is that the features are attached to a non-involved class. As a consequence, the characteristic `totalBalance` represents a set of links between debts and data values of type Euro instead of a set ternary links between persons, banks and data values of type Euro. It is more than likely that a certain person has accounts at a certain bank but that this person has no debts (at this bank). In such case, the features

`totalDeposit()` and `totalBalance` do not exist for the given person and bank. Obviously, this is unwanted.



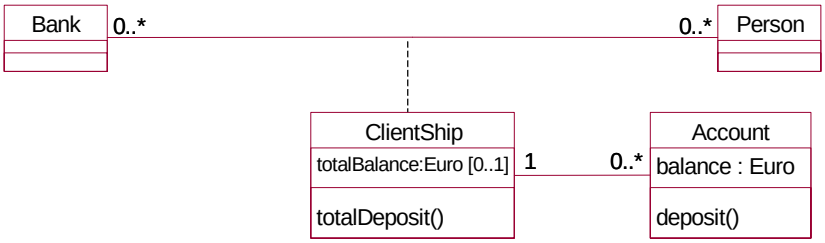
context Account :: deposit(amount: Euro)
post: balance() = balance()@pre + amount

context Debt :: totalDeposit(amount: Euro)
post: bank().account()→intersection(person().account()).deposit(amount)
-- *person().account()→intersection(bank().account()).deposit(amount)*

context Debt **inv:** totalBalance() =
bank().account()→intersection(person().account()).balance()→sum()
-- *person().account()→intersection(bank().account()).balance()→sum()*

Figure 4 Attaching the features to the class Debt

An association class `Clientship` is added to the conceptual model of Figure 5. The class represents the set of clientship links. A person can be a client at a certain bank. Furthermore, it is assumed that a person cannot have accounts at a bank without being a client at that bank. This model violates the principle of abstraction. An analyst is forced to further investigate the real world in order to find a possible class to attach the features. Another important objection is that the features are attached to a non-involved object. If the real-world changes and a person must not be a client before opening an account, the same problems as in the previous model occur. Furthermore, it will sometimes impossible to find such class `Clientship`. Consequently, such an alternative is not a general solution to the problem.



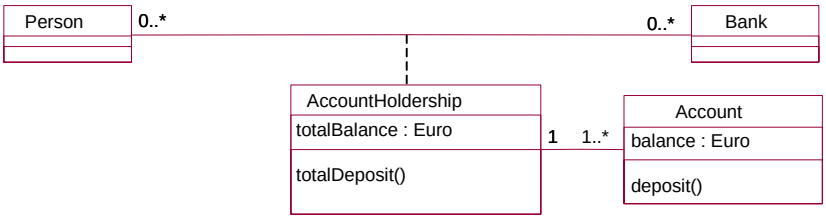
context Account :: deposit(amount: Euro)
post: balance() = balance()@pre + amount

context Clientship :: totalDeposit(amount: Euro)
post: account().deposit(amount)

context Clientship **inv:** totalBalance() = account()→balance()→sum()

Figure 5 Attaching the features to the class Clientship

In the conceptual model of Figure 6, the association class `AccountHoldership` is introduced. An accountholdership link between a given bank and given person exists if the given person has at least one account at the given bank. The principle of abstraction is not violated, but we believe that the introduction of the class `AccountHoldership` is artificial. If a person opens an account there is "automatically" an accountholdership. This makes the specification of the opening of an account more complicated. When an account is opened an accountholdership must be created or must already exist. Should there be a different specification for opening the first account and for opening additional accounts? Another disadvantage is that the features are attached to a non-involved class. The characteristic `totalBalance` represents a set of links between accountholderships and data values of type Euro and not between persons, banks and data values of type Euro. The ternary links between persons, banks and data values of type Euro are not explicitly represented. No statements about the characteristic are made for persons that have no accounts at a certain bank.



context Account :: deposit(amount: Euro)
post: balance() = balance()@pre + amount

context AccountHoldership :: totalDeposit(amount: Euro)
post: account().deposit(amount)

context AccountHoldership **inv:** totalBalance() = account()→balance()→sum()

Figure 6 Attaching the features to the class AccountHoldership

Notice, that in the model of Figure 6 the ternary relationship underlying the static property "totalBalance" is in fact split up into two binary relationships: one between banks and persons, called accountholdership, and one between accountholdership and a data value of type Euro.

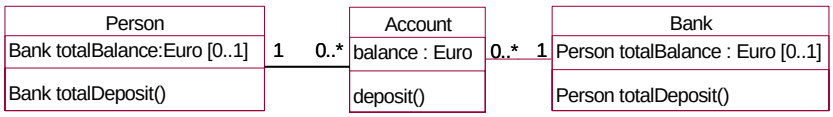
1.3 Patterns

1.3.1 N-ary Features

In order to model properties in which several objects are involved, the construct of ***n-ary features*** is introduced. We call such features binary features if two objects are involved; ternary features if three objects are involved… The general name for a feature involving n objects is n-ary features. N-ary features are attached to all n involved classes. Consider as an example the binary features `totalDeposit()` and `totalBalance` in Figure 7.

Aside the introduction of a new construct, the use of objects as explicit arguments in events is forbidden. Classes as type for a characteristic are also

forbidden. This prohibition guides the analyst in representing the real-world observations and avoids construct redundancy.



context Bank,Person :: totalDeposit(amount: Euro)
post: (selfBank.account()),(selfPerson.account()).intersection().deposit(amount)

context Bank,Person **inv:** totalBalance() =
(selfBank.account()),(selfPerson.account()).intersection().balance()→sum()

Figure 7 N-ary features

The introduction of n-ary features has some notational consequences:

- Graphically, n-ary features are modelled in all the classes involved. In order to emphasize this n-ary aspect, the name of the other classes involved in an n-ary message is written before the name of the feature.

- An n-ary feature can be applied on n objects. A comma separates the involved objects. Notice that the feature `intersection` can also be interpreted as a binary feature involving two sets. The application of the binary feature `intersection` is illustrated in the specification of the postcondition and the invariant.

- The implicit arguments are referenced by "self" followed by the name of the class. If more than one object of the same class is involved the implicit arguments are denoted self1Class, self2Class and so on. If no other classes are involved we could abbreviate to self1, self2 and so on.

More important is that also the specification for the postcondition and the invariant is changed. Instead of one context class, we now have two or more context classes. Instead of starting with one object and navigating through the model, the postcondition and the invariant have now two starting objects. No choice between starting objects must be made. Moreover, no long

navigation from one object to another object is needed. This way of specifying is more declarative and more neutral in the sense that it doesn't suggest an implementation strategy. In general, n-ary features allow making abstraction of classes or associations. Notice that also an invariant can have more than one context.

A typical application of an n-ary feature is the linking of two or more objects. Consider the association among persons and cars in Figure 8. It is assumed that several persons can be the owner of the same car. How should the acquisition of a car by a person be reflected in terms of messages? Is a person buying a car? Or is a car bought by a person? Should there be a feature attached to a class `Person`, to a class `Car` or to both classes? When using n-ary features the answer is obvious: the process of buying a car becomes a binary event involving both a person and a car.

context Person,Car :: buy()
post: selfPerson.car() = selfPerson.car()@pre.includes(selfCar)
-- *selfCar.person() = selfPerson.car()@pre.includes(selfPerson)*

Figure 8 A typical application of n-ary features

Notice that there are still problems in specifying the effect of the event `buy()`. As illustrated in Figure 8, several alternatives exist to specify the effect of the event. The possibility of alternative specifications is caused by the given implicit queries, and not by the binary feature itself. Moreover, these alternatives would still exist if the event had been only attached to the class `Person` or to the class `Car`.

Notice that the association between the class `Person` and the class `Car` could be replaced by a binary characteristic `ownership: Boolean` attached to the classes `Person` and `Car`. Further evaluation about the use of both constructs is needed.

1.3.2 Further Research

1.3.2.1 Ambiguity during Specialization / Generalization

Some amount of ambiguity arises when n-ary features become involved in hierarchies of classes. Consider the class-diagram in Figure 9. Two binary events `m()` with different contexts are modelled. If the message `m()` is applied to the object `a1` of class `A1` and to the object `b1` of the class `B1`, the selection between the two given binary messages is ambiguous. This kind of ambiguity is also described in [Schr91, p.303], where cooperation contracts, which can be more or less compared with n-ary features, are proposed for object-oriented database design. Cooperation contracts are said to offer advantages such as symmetricity, locality, extendibility and maintainability [Schr91, p. 296].

context A,B1 :: m()
post: …

context A1,B :: m()
post: …

Figure 9 Ambiguity during specialization / generalization

Some possible solutions are briefly proposed. The simplest one is to forbid ambiguity. Another could be the obligation to introduce a binary event `m()` with contexts A1 and B1. A more complex one is to combine the specification of both events `m()` through logical conjunction.

1.3.2.2 Queries and the Message Paradigm

Queries and events are applied on objects. In object-orientation, it is said that messages are send to objects. This principle is known as the ***message paradigm*** [Devo01, p.183]. The message paradigm is, apart from inheritance, one of the most specific constructs in object-orientation. The receiving object(s) is the sole responsible for responding the message. The receiving object may appeal to other objects by sending them messages in turn. The receiving object is the implicit argument for the underlying message.

In [Stee99, p.48] it is argued that the message paradigm simulates the way people interact with all kinds of things in their surroundings. When changing the channel on a television, a message (signal) is sent by means of a remote control. In responding, the television will sent proper messages to some of its components. This explains why the message paradigm is said to be a natural way of communicating. In [Jack95, p.137], Jackson questions the idea of sending messages to objects for analysis purposes. He argues that it makes no sense to send messages to real-world objects as bottles, trees and pay checks. We do not adopt the semantics of sending and receiving messages to and from real-world objects during analysis. In other and more complex words, ***anthropomorphism*** [Brow02, p.137], or the process of giving human qualities and abilities to nonhuman objects is not supported.

Reconsider the characteristic `totalBalance` in Figure 7. This characteristic represents a set of links between persons, banks and data values of type Euro. On top of this representation different information requirements can be needed. Until now, only an implicit query `totalBalance()` is given. This query can be applied on a given bank and a given person. It returns a data value of type Euro. Other kinds of information requirements could be equally important. Perhaps a more neutral kind of queries should be introduced. For an n-ary characteristic with m data types, an implicit query `(class1,…, ClassN) characteristicName(DataType1,…,DataTypeM)` could be implicitly given. The desired output of the query is denoted with a question mark. The input of the query can be objects *and* data values. Consider the conceptual model in Figure 10. The query `(myAccount)limit(?)` returns the limit of my account. The query

`(?)limit(500 Euro)` returns the set of accounts that have a limit of 500 Euro. The query `(Fortis,Jan) totalBalance(12/09/2003,?)` returns the total balance of Jan at the Fortis bank on 12 September 2003. The query `(Fortis,?) totalBalance(?,1.000.000 Euro)` returns a set of persons and dates. These persons had a total balance of 1.000.000 Euro at the Fortis bank on the indicated date.

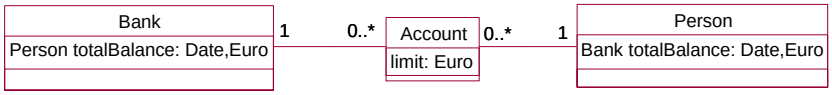


Figure 10 Implicit queries for characteristics

The same kind of queries could be offered for associations. For an n-ary association an implicit query `(Class1,…,ClassN)associationName (Boolean)` is given. Consider the example in Figure 11. A query `(Jan,?)ownership(true)` returns all the cars of the Jan. A query `(Jan,thisCar)ownerschip(?)` returns if Jan is the owner of this car. As illustrated in the postcondition of Figure 11, this kind of queries would solve the problem of several possible alternatives for the specification of the effect of the event `buy()`.

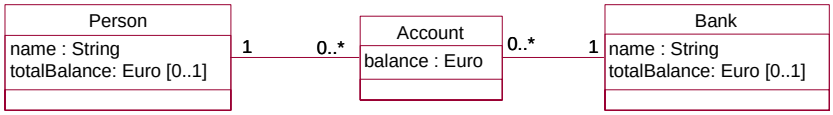
context Person,Car :: buy()
post: (selfPerson,selfCar)ownership(?)

Figure 11 Implicit queries for associations

This kind of implicit queries seems to be more powerful and more neutral in relation with the possible information requirements. In such a case, queries are not longer only applied on objects. Further evaluation is however needed.

1.3.2.3 Or-context Features

N-ary features are and-context features: at least two objects are involved. If and-context messages are allowed, the question rises if or-context messages could also be useful. In such a feature an object of a certain set or of another set is involved. Consider the features `totalBalance` of the classes `Person` and `Bank` in Figure 12. The total balance of a person equals the sum of the balances of all his accounts. The total balance of a bank equals the sum of the balances of all its accounts. Both features have the same specification. In the second alternative, this is modelled with the help of an or-context. The or-context is denoted by a |. The advantage is clear: the analyst must specify the dependency only once.



Alternative One *-- no or-context*
context Bank **inv:** totalBalance() = account().balance()→sum()
context Person **inv:** totalBalance() = account().balance()→sum()

Alternative Two *-- or-context*
context Bank | Person **inv**: totalBalance() = account().balance()→sum()

Figure 12 Features with an or-context

Generalizing the class `Person` and `Bank` could also reach this advantage but this kind of generalization is rather rarely observed. In [Beka01, p.34], we argued that the driving force in object-oriented analysis to use the construct of generalization is polymorphism: the capability and need to reason about the union of sets and the possibility to refine the reasoning for objects of the specialization classes, and not the fact that classes have common features. Generalization is further discussed in chapter 6.

1.3.3 Conclusion

The construct of n-ary features is introduced and allows the analyst to attach a feature to more than one class. Features are attached to all classes involved. As for associations, no distinction is made in attaching a feature to classes. This construct promotes abstraction and eliminates arbitrary choices. Moreover, n-ary features lead to a more appropriate specification of events and invariants.

2 Features involving no Objects or artificial Objects

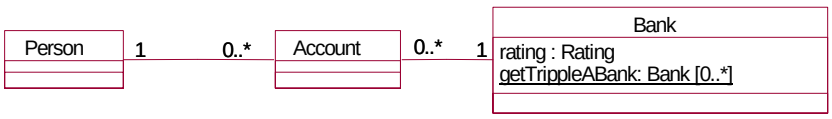
2.1 Problem Description

Another problem with features arise when no or only artificial objects are involved. This could be the case when a subset of objects, based on some certain static property, must be represented.

2.2 Anti-Patterns

2.2.1 Class-scoped Features

Class-scoped features [OMG01, p. 3-43 and p.3-45] can be used when no specific objects are involved. In the UML, this kind of features is underlined. Consider the example in Figure 13. The class-scoped feature `getTrippleABank` represents all banks with an "AAA" rating. Notice that we could have attached the invariant or the feature to no matter which class in the conceptual model.

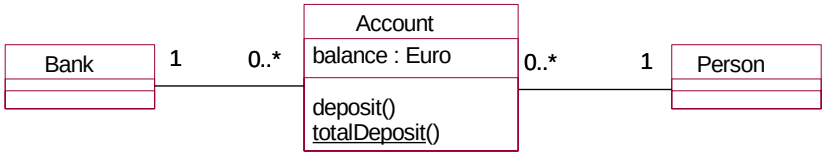


context Bank **inv:** Bank.getTripleABank() = Bank.AllInstances→select(rating() = "AAA")

Figure 13 Class-scoped features

A problem with class-scoped features is that they can be misused to model object-scoped features. This is frequently observed in practice. The use of class-scoped events could, in an extreme way, lead to functional decomposition or structured design. In functional decomposition the system function is organised as a hierarchy of smaller functions, and the development consists of elaborating this hierarchy from the top downwards [Jack83, p.4].

Consider as an example the class diagram in Figure 14. The object-scoped event `totalDeposit()` of the previous section is modelled as a class-scoped event where the two explicit arguments are objects. Notice that this class-scoped event could be attached to any possible class. Consequently, an adequate classification of features when using the class-scoped mechanism is lost. Therefore, the construct of class-scoped features will not be offered to the analyst.



context Account :: deposit(amount: Euro)
post: balance() = balance()@pre + amount

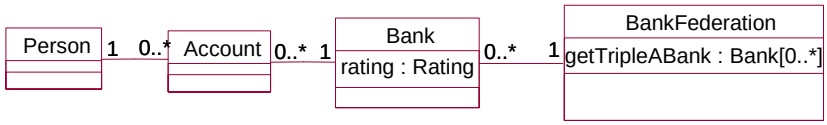
context Account :: totalDeposit(person:Person, bank:Bank, amount: Euro)
post: Account.AllInstances→select(person() = person and bank() = bank).
deposit (amount)

Figure 14 Class-scoped features

2.2.2 Introducing Artificial Classes

Another, more object-oriented way to avoid class-scoped features is to introduce a new class with object-scoped features. In the banking example, one could introduce the class `BankFederation`. In that context, we assume

that a bank is always linked with a bank federation. Consider the class-diagram in Figure 15. The object-scoped characteristic `getTripleABank` selecting all banks with an "AAA" rating is attached to the new class `BankFederation`.



context BankFederation **inv:** getTripleABank () = bank()→select(rating() = "AAA")

Figure 15 Introducing a new class

This approach, although more object-oriented, has several disadvantages:

- The first disadvantage is that an artificial class that does not belong to the problem context is introduced. The only reason why this class is introduced is that it permits the analyst to specify a feature that represents a selection of objects. This is a technical reason. In our opinion, only classes that play an important role in the problem context should be modelled.

- Furthermore, it is assumed that there is only one bank federation object. Only then is it possible to select all the banks with a specific property. The need for class-scoped features is also postponed. How could the bank federation object be selected? By introducing a new class? Or by introducing a class-scoped feature `getBankFederation()`?

- If this approach would be adopted, then new singleton classes should be introduced for other selection of objects. In the example, a new artificial class should be introduced for characteristics representing a selection of persons.

- Another problem is that this approach could lead to long navigation expressions. Suppose that a selection of all transactions on accounts on a certain date should be modelled. This message could be attached to the

class `BankFederation`. From a bank federation we are navigating to all the banks, and from there to all the accounts and at last to all the transactions on accounts.

- Characteristics are reserved to model sets of links between data values and objects. The feature `getTripleABank` is a characteristic but models a set of links between a bank federation and triple "A" banks. For this kind of links, an association should be used. The use of characteristics to model links between objects is not allowed. Furthermore, if triple "A" banks have special dynamic or static properties, these properties cannot be attached to an already existing class.

2.3 Patterns

2.3.1 Subset Constraints

To resolve the disadvantages of the previous approach, (sub)sets of objects will be modelled as classes and the construct of a ***subset constraint*** will be introduced. In the UML, some predefined generalization constraints [OMG01, p.3-87] such as `overlapping`, `disjoint`, `complete` and `incomplete` already exist. A generalization constraint indicates a dependency among the subclasses that share a common superclass. A subset constraint indicates a dependency between the superclass and the subclass.

Consider the class-diagram in Figure 16. The set of all triple "A" banks is now represented by a class. The subset constraint specifies that all the banks with an "AAA" rating are also objects of the class `TripleABank` and that all objects of this class have an "AAA" rating. Notice that an invariant attached to the class `TripleABank`, specifying that all objects of this class have a triple "A" rating is not a valid alternative for the subset constraint. The invariant would be too weak. This approach resolves the disadvantages of the previous approach.

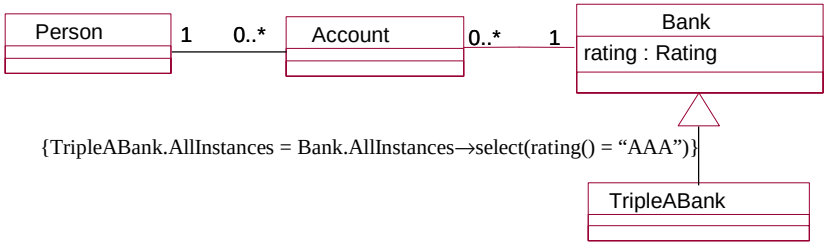
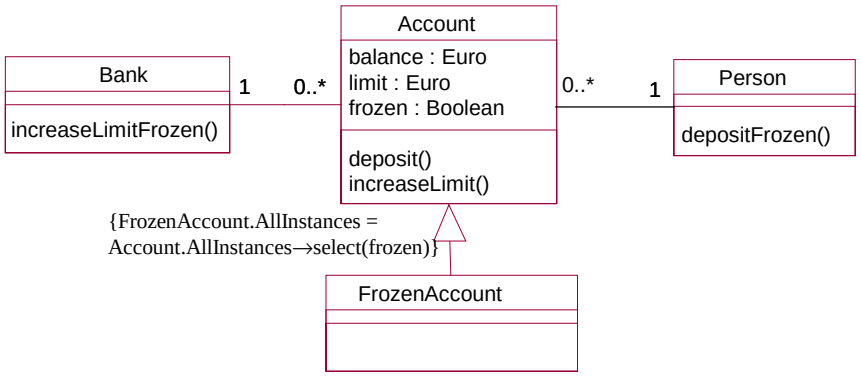


Figure 16 Subset constraints

Another advantage is that the same selection of subsets must be specified only once. In other words, the multiple use of the selection statement in the specification of events or invariants can be avoided. Consider an example in Figure 17. Assume a class `FrozenAccount`, representing a certain subset of all accounts. In the specification of the event `depositFrozen()` attached to the class `Person`, the select statement is not used. The implicit query `frozenAccount()` is used instead. This event increases the balance of all frozen accounts of a given person with a given amount. The same selection statement is avoided in the event `increaseLimitFrozen()`.



context Account :: deposit(amount: Euro)
post: balance() = balance()@pre + amount

context Account :: increaseLimit(percentage: Real)
post: limit() = limit()@pre (1 + percentage)

context Person :: depositFrozen(amount: Euro)
post: frozenAccount().deposit(amount)
-- *account()→select(frozen).deposit(amount)*

context Bank: increaseLimitFrozen(percentage: Real)
post: frozenAccount().increaseLimit(percentage)
-- *account()→select(frozen).increaseLimit(percentage)*

Figure 17 Avoiding select statements

2.3.2 Parameterised Classes

Not only the construct of a subset constraint should be introduced but also the construct of a ***parameterised class*** or template class is necessary to represent possible subsets of objects. A template class defines a family of classes, each class specified by binding its parameters to actual values [OMG01, p.3-52]. Consider the example outlined in Figure 16. The subset of banks with a certain rating is represented with the help of a parameterised class. The parameter is used in the subset constraint.

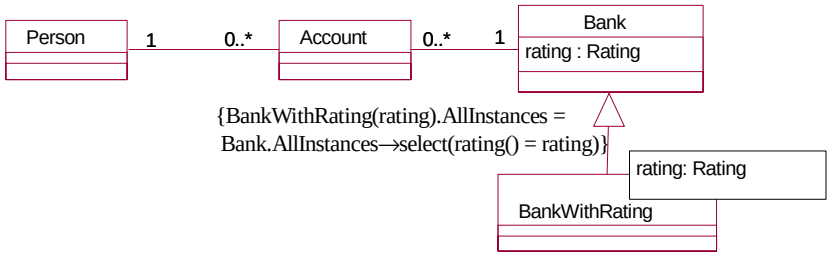


Figure 18 Parametrized classes

One could argue that this pattern does not give an answer on how to model for instance the average rating of all banks or the change of a rating of all or

a subset of banks. This is correct but is discussed in a section about the scope of the conceptual model in the last but one chapter.

2.3.3 Conclusion

Subsets of objects, based on a certain static property, should be modelled as subclasses. The construct of subset constraints and parametric classes are introduced in order to formally specify the relationship between the superclass and the subclass. Due to this pattern, the use of class-scoped messages or the introduction of artificial singleton classes can be avoided. Furthermore, the use of characteristics is reserved for the representation of links between objects and data values and cannot be used to model links between objects.

Chapter 5

Patterns for Modelling Restrictions

1 Introduction

1.1 Restrictions

In the external world, an analyst observes human or business-imposed, physical or legal laws, rules and regulations. These ***restrictions*** delimit certain dynamic properties and the evolution of static properties [Steeg00, p.95]. Restrictions are real-world facts. In the case of enterprise modelling most of these restrictions reflect business rules.

It is surprising to observe that many object-oriented analysis methods [Booc94][Coad90][Jaco93][Larm98][Peet01, p.88] do not emphasize the importance of restrictions. A constraint in these methods is considered a detail and not a major element in developing conceptual models. Frequently, restrictions are not articulated until it is time to convert them into program code [Guid97]. Aside sets of objects, static and dynamic properties, we classify restrictions as a fourth eminent type of real-world facts to represent. The ability to represent restrictions properly is essential to successful conceptual modelling. Remind that one of principles of conceptual modelling is to ensure the completeness of the conceptual model.

The UML provides the Object Constraint Language (OCL) to model restrictions in a formal way [OMG01, p.6-2]. This constraint language is based on first-order logic and set theory. The OCL provides constraints such as preconditions, postconditions and invariants as (traditional) constructs for representing restrictions [OMG01, p.6-5].

Notice that the use of invariants, pre- and postconditions during the analysis phase suggests in no way an implementation strategy [OMG01, p.6-1]. Defensive programming using exceptions, total programming using

extended postconditions and nominal programming using preconditions are still all possible at the implementation level.

1.2 Constraints during Analysis and Design

During analysis, an invariant is defined as a condition that must be true at each moment in time, without determining *how* and *when* this condition must be controlled [Gogol98, p.110] [Vanb93, p.393]. Because of the declarative nature of analysis, in which events are assumed to be instantaneous, this timeless definition of an invariant is acceptable.

Due to its operational nature, the definition of an invariant during the design phase must be less restrictive: an invariant must be true at all times, except during the execution of an operation. Warmer and Kleppe, the main authors of the OCL, seem to adopt this design view on an invariant. Their initial statement that “an invariant must be true all the time” [Warm99, p.4] has been rephrased as "the invariant must be true upon completion of the constructor and every public method but not necessarily during the execution of methods" [Klas02].

In [Henn02, p.72], an additional tag for non-public operations is suggested. In particular, the tagged value `{volatile=true}` can be attached to private and protected operations indicating that invariants should not be satisfied at the end of their execution. In case of absence of this tagged value, the validity of the invariants is undefined for non-public operations. Notice that this tag is not selective enough to specify which invariants hold at the end of a non-public method, and which invariants do not hold. More fine-grained constructs could be necessary during design.

As for invariants, different semantics are also ascribed to preconditions imposing restrictions to individual operations. At the level of analysis, a precondition states the conditions to be satisfied each time the event occurs. As for invariants, there is no need to focus on the precise moment preconditions are checked. At the level of design, preconditions are often linked to exception handling, in the sense that exceptions are thrown if upon entry to a method some precondition turns out to be violated [Meye97, p.413]. A similar reasoning could be made for postconditions.

The different definitions of the construct of invariants, preconditions and postconditions illustrate the need for different constructs during analysis and design. This need does not obstruct a smooth transition from analysis to design. Both phases have different goals. Because of these different goals, different constructs are needed (or different semantics for similar concepts).

In section 2, the way to specify restrictions on static properties in combination with specification of the effect of an event is discussed. In section 3, the way to specify restrictions on dynamic properties is described. In these sections, the constructs of invariants, preconditions, implications and postconditions are evaluated. In section 4, the transactional semantics of events is discussed. In section 5, the frame axiom in case a constraint is about to be violated is relaxed. Section 6 evaluates implicitly versus explicitly modelled class constraints.

2 Modelling Restrictions on Static Properties

2.1 Problem Description

Dependencies between static properties are restrictions on static properties. How can such a restriction be adequately specified in combination with the specification of the effect of events? Consider the example in Figure 1. Assume that the balance of account must always exceed its limit. A withdrawal decreases the balance of an account with a given amount. The specification of the effect of the event can possibly violate the specification of the business rule. How do we specify the business rule and the effect of the event?

Figure 1 Modelling restrictions on static properties

Three different approaches are evaluated. In the first approach only an invariant is used to model the restriction. In the second approach, combinations of an invariant and preconditions or implications are used to represent a real-world business rule. In the last approach, a new construct is introduced to describe the business rules.

2.2 Anti-Patterns

2.2.1 Invariants

In this first approach, an invariant is used to model a restriction imposed on a set of real-world objects. An invariant at the level of analysis must be satisfied at any time. Consider the example outlined in Figure 2. An invariant is used reflecting a business rule stating that the balance of an account must exceed its limit.

Problems arise as soon as events must be specified. In the OCL, events are specified with the help of preconditions and postconditions. The principle behind these conditions is the ***contract paradigm*** [Warm99, p.2]. In a contract, the obligations and rights between the different parties are stipulated. This contract paradigm guarantees that if the precondition holds, the postcondition of the event is satisfied after the occurrence of the event. Consider the specification of the event `withdraw()`. This specification states that, upon occurrence, the balance of the account will be diminished with the given amount. In the next paragraphs, it will be demonstrated that in the example the contract principle is broken. In particular, we will show that the postcondition of the stated event can be violated under some conditions.

context Account **inv**: balance() >= limit()

context Account :: withdraw(amount : Euro)

post: balance() = balance()@pre – amount

Explicated specification of the event
context Account :: withdraw(amount : Euro)
epre: true
epost: self.balance() = self.balance()@pre – amount and
Account.AllInstances→forAll(balance() >= limit()) and --*invariant*
Account.AllInstances→forAll(limit() = limit()@pre) and --*frame axioms*
Account.AllInstances→excluding(self)→forAll(balance() = balance()@pre) and
Account.AllInstances= Account.AllInstances@pre

Figure 2 Invariants

In the explicated specification of the event, the implicit assertions about the event are made explicit. These explicated assertions help to demonstrate that the contract principle is broken. In absence of a precondition, events have `true` as their precondition [Meye97, p.362]. This is illustrated in the explicated precondition (epre). Because invariants are always true, invariants must also be true after the occurrence of an event. This is illustrated in the second assertion of the explicated postcondition (epost).

Notice that the frame axiom is adopted: a modification must be specified explicitly in the postcondition; everything else is left untouched [Borg95]. Consequently, a withdrawal leaves the limit of an account untouched and cannot be modified in order to respect the invariant. The last three assertions in the postcondition make the frame axiom explicit.

Reconsider the explicated postcondition. In case too much money is withdrawn from an account, the second assertion contradicts the first assertion. In this case, the postcondition cannot be guaranteed even if the precondition holds. Consequently, the contract principle is broken. Obviously, the problem of the broken contract principle persists in case of invariants at the level of specialization.

2.2.2 Invariants and Preconditions or Implications

A possible approach to respect the contract principle consists in eliminating the contradiction by means of a more detailed specification of the event.

Consider the example in Figure 3. In this approach, the specification of the event restricts the conditions under which the effect of the events can occur. An implication is used in the first alternative, while a precondition is applied in the second. There are no situations where the explicit assertion in the postcondition contradicts the invariant. Given that the precondition holds, the (implicit and explicit) assertions in the postcondition do not contradict each other. Consequently, the contract principle is not broken. The explicated specifications of the events are not given but could be easily derived from the implicit specifications.

context Account **inv**: balance() >= limit()

Specification with an implication
context Account :: withdraw(amount : Euro)
post: (balance()@pre – amount >= limit()) implies balance() = balance()@pre – amount

Specification with a precondition
context Account :: withdraw(amount : Euro)
pre: balance() - amount >= limit
post: balance() = balance()@pre – amount

Figure 3 Invariants and preconditions or implications

The extra specifications, necessary to avoid the problem of the broken contract principle, violate the principle of modelling a real-world fact only once. If the specification of n events can possibly contradict the invariant, the real-world restriction is represented in n+1 places: once as an invariant and n times as a precondition or an implication in the specification of each event. As stated in [Steeg00, p.95], much too often software engineers are focusing from the very start on *how* and *when* to check potential constraints violations. This approach partially forces the software engineers to do so.

Because the same business rule occurs at different points in the conceptual model, this approach does not support adaptability and extendibility.

Adaptability is decreased, because changing the business rule means changing the invariant and changing the specification of events. Extendibility is decreased, because adding an invariant may require adding conditions in the specification of some events.

With this approach, problems also arise in case new invariants are added or existing invariants are strengthened at the level of specialized classes. New invariants can be added or existing invariants can be strengthened [Dock99, p.316] without violating ***the substitution principle of Liskov*** [Lisk94]. This principle states that whenever an object of a class is expected, one can always substitute that object by an object of any of its subclasses. According to the same principle, preconditions can be weakened at the level of specialized events [Dock99, p.323]. Similar problems during specialization-generalization can be observed when implications are used instead of preconditions.

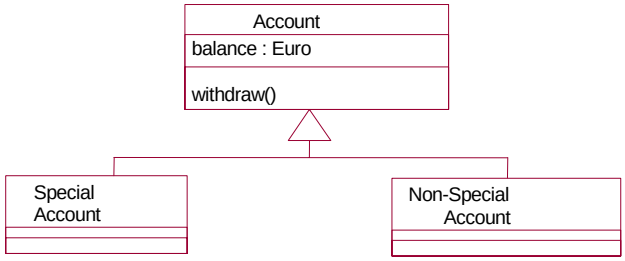
The problems raised by adding or strengthening invariants at the level of subclasses are illustrated in the example outlined in Figure 4. Assume a real-world situation in which accounts and special accounts have a balance and where withdrawals occur. Special accounts are a subset of accounts. A business rule states that the balance of each account amounts to at least 1000 Euro. Another business rule states that the balance of each special account amounts to at least 2000 Euro. In this example there is no attribute `limit`. The reason is that we want to use a very simple example that focuses on the difficulties of invariants and preconditions at the level of specialization. A more complex example could be envisaged and would reveal the same problems.

In Figure 4, the approach with preconditions reveals three problems:

- Firstly, the specification is contra-intuitive: at the level of the generalization the invariant uses 1000 Euro in its specification while the precondition uses 2000 Euro.

- Secondly, no subclass of the class `Account` can be added that strengthens the invariant with more than 2000 Euro without the need to change the existing precondition of the event `withdraw()` of this class.

- Thirdly, no subclass of the class `Special Account` or of the class `Non-special Account` can be added that strengthens the invariant without the need to change the existing preconditions of these classes.



context Account **inv**: balance >= 1000 Euro

context Account :: withdraw(amount : Euro)
pre: balance – amount >= 2000 Euro
post: balance = balance@pre – amount

context Special Account **inv**: balance >= 2000 Euro

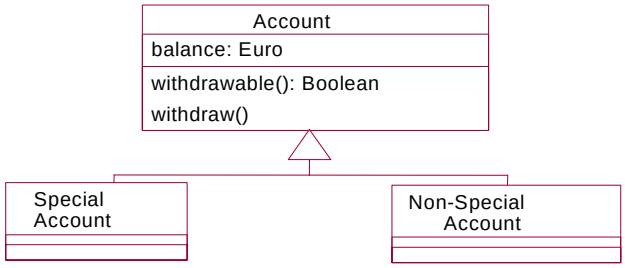
context Non-Special Account :: withdraw(amount : Euro)
pre: balance – amount >= 1000 Euro

Figure 4 Specializing preconditions and invariants

Notice that the class `Non-Special Account` and the precondition of the event `withdraw()` of this class is necessary to guarantee the effect of a withdrawal for non-special accounts in case their balance minus the amount of the withdrawal is situated between 1000 and 2000 Euro. Indeed, the specification of the event at the level of the general class of accounts does not guarantee the applicability of such withdrawals.

A possible approach to avoid some of the three problems mentioned above is the use of abstract preconditions [Meye97, p.577]. An abstract precondition is a precondition with a property that is redefinable at the level of specialized classes. The advantage of abstract preconditions is illustrated in Figure 5. The conditions under which an amount of money can be withdrawn from an

account are encapsulated in a Boolean inspector `withdrawable()`. The postcondition of this inspector is strengthened at the level of the class `Special Account` and of the class `Non-Special Account`. This approach resolves only the first two problems that arise in case non-abstract preconditions are used. Consequently, the invariants of the class `Special Account` or of the class `Non-Special Account` cannot be further strengthened without the need to change the existing specification. Furthermore, the approach with abstract preconditions does also not resolve the general extendibility and adaptability problems that are described in the second paragraph of this subsection. Notice, that the class `Non-Special Account` is also necessary in this approach.



context Account **inv**: balance() >= 1000 Euro

context Account :: withdraw(amount : Euro)
pre: withdrawable(amount)
post: balance() = balance()@pre – amount

context Account :: withdrawable(amount : Euro): Boolean
post: **(**balance – amount < 1000 Euro) implies result = false

context Special Account **inv**: balance() >= 2000 Euro

context Special Account :: withdrawable(amount : Euro) : Boolean
post: result = balance() – amount >= 2000 Euro

context Non-Special Account :: withdrawable(amount : Euro): Boolean
post: result = balance() – amount >= 1000 Euro

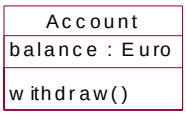
Figure 5 Specializing abstract preconditions and invariants

2.3 Patterns

2.3.1 Class Constraints and the Principle of Non-Violation

A third approach consists in modelling general business rules in terms of ***class constraints*** [Steeg00, p.95]. As for invariants, class constraints must be true at each moment in time for all objects of the class, but the semantics of a class constraint are complemented with the ***principle of non-violation***. This principle states that occurrences of events that would violate a class constraint are simply rejected. If there are no class constraints that would be violated, the event succeeds.

A class constraint is worked out in Figure 6. It states that the balance of each account must at all times exceed its limit. In view of events, we rely on the principle of non-violation. This principle states that occurrences of events that would violate a class constraint are simply rejected. If there are no class constraints that would be violated, the event succeeds. As an immediate consequence, the state of all objects populating a conceptual model is left untouched each time an event occurs that would violate some business rule. Consider the explicated specification of the event `withdraw()`. This specification is now complemented with an implicit assertion stating that the balance of the involved account does not change each time an attempt is made to withdraw too much money.



context Account **constraint**: balance() >= 1000 Euro

context Account :: withdraw(amount : Euro)
post: balance() = balance()@pre – amount

Explicated Specification of the event
context Account :: withdraw(amount : Euro)
epost: if (balance@pre – amount >= 1000 Euro)
then balance() = balance()@pre – amount
else balance() = balance()@pre endif and

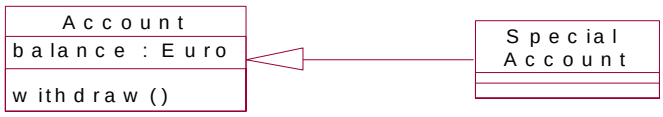
```
Account.AllInstances→forAll(balance >= 1000 Euro) and    --class constraint
Account.AllInstances= Account.AllInstances@pre  and      --frame axiom
Account.AllInstances→excluding(self)→forAll(balance = balance@pre)
```

Figure 6 Class constraints

This approach supports adaptability and extendibility of the model. When new class constraints are added to the model or existing class constraints are changed, there is no need to modify the explicit specification of events. New events can also be easily added to the model without examining all possible conditions under which they may violate class constraints. For the same reason, the specification of existing events is easily changed.

2.3.2 Class Constraints at the level of Specialization

This approach also fully supports the strengthening of class constraints at the level of specialized classes. Assume the same situation with special accounts as in the previous approach and consider Figure 7. In the approach with class constraints, the class constraint in the class `Special Account` can be further strengthened without the need to change existing specifications of events or other elements of the conceptual model. Furthermore, there is no need for the class `Non-Special Account`.



context Account **constraint**: balance >= 1000 Euro
context SpecialAccount **constraint**: balance >= 2000 Euro

context Account :: withdraw(amount : Euro)
post: balance = balance@pre – amount

Figure 7 Class constraints at the level of specialization

Consider the explicated specification of the withdraw-event for accounts and special accounts in Figure 8. If the condition (`balance@pre – amount <`

1000 Euro) is satisfied, the event fails for all accounts and the state of the model remains unchanged. If the condition (balance@pre – amount >= 2000 Euro) is satisfied, the event succeeds for all accounts and the balance of the account is decreased with the given amount. In the other cases, the event can fail or can succeed depending on the type of account. In the example hierarchy, the withdrawal on an account, which is not a special account, is guaranteed if the condition (balance@pre – amount >= 1000 Euro and balance@pre – amount < 2000 Euro) is satisfied. In such conditions, a withdrawal on a special account is rejected and the state of the model is left untouched.

Explicated specification of the event for accounts

context Account :: withdraw(amount : Euro)
epost: (balance@pre – amount < 1000 Euro implies balance = balance@pre) and
(balance@pre – amount >= 2000 Euro implies balance = balance@pre – amount) and
(balance@pre – amount >= 1000 Euro and balance@pre – amount < 2000 Euro and
not (self.oclIsKindOf(SpecialAccount) implies balance = balance@pre – amount) and

Account.AllInstances→forAll(balance >= 1000 Euro) and --*class constraint*
Account.AllInstances= Account.AllInstances@pre and --*frame axioms*
Account.AllInstances→excluding(self)→forAll(balance = balance@pre)

Explicated specification of the event for special accounts

context SpecialAccount :: withdraw(amount : Euro)
epost: (balance@pre – amount < 2000 Euro implies balance = balance@pre) and
(balance@pre – amount >= 2000 Euro implies balance = balance@pre – amount) and

SpecialAccount.AllInstances→forAll(balance >= 1000 Euro) and --*class constraint*
SpecialAccount.AllInstances= SpecialAccount.AllInstances@pre and --*frame axiom*
SpecialAccount.AllInstances→excluding(self)→forAll(balance = balance@pre)

Figure 8 Explicated specification for accounts and special accounts

2.3.3 Generalizing the explicated Specifications

The relation between the shortcut specification for an event and the more extended specification is generalized in Figure 9. The frame axiom and the principle of non-violation are both taken into account in the extended

version. The assertion `frame()` indicates that the state of the model is unchanged. The assertion `frame()\event` is the conjunction of all frame axioms in case an event succeeds. The assertion `classConstraints()` is the conjunction of all class constraints and must always evaluate to true. An assertion with a subscript $_x$ renames all properties of this assertion without a subscript with a subscript $_x$. For instance, the subscript $_{pot}$ denotes that all properties are replaced by "potential" properties. The state of a potential property is the same as the state of the property of the model before the occurrence of the event unless otherwise specified. The assertion `not(assertion()`$_{pot}$ `and frame()\event`$_{pot}$ `and classConstraints()`$_{pot}$`))` is added in the extended version of the postcondition because the occurrence of an event leaves the state of the model unchanged if and only if at least one class constraint is about to fail. If this assertion would not be added, the semantics of the event would be that it is always possible that occurrences of the event would leave the state of the model unchanged. This is not the case. In the second explicated version, the assertion `succeed(event)` is and abbreviation for `(assertion() and frame()\event() and classConstraints())`. The assertion `fail(event)` is an abbreviation for `(frame() and classConstraints() and not( assertion()`$_{pot}$ `and frame()\event`$_{pot}$ `and classConstraints()`$_{pot}$`))`.

In General

context Class :: event() -- *Shortcut version*
post: assertion()

context Class :: event() -- *Explicated version (1)*
epost: (assertion() and frame()\event and classConstraints()) **or**
(frame() and classConstraints() and not(assertion()$_{pot}$ and frame()\event$_{pot}$ and classConstraints()$_{pot}$))

context Class :: event() -- *Explicated version (2)*
epost: succeed(event) **or**
(frame() and fail(event))

An Example

context Account :: withdraw(amount: Euro) -- *Shortcut version*
post: balance() = balance()@pre - amount

context Account **constraint:** balance() >= limit()

context Account :: withdraw(amount: Euro) --*Explicated version*
epost: (balance() = balance()@pre – amount and limit() = limit()@pre and balance() >= limit()) **or**
((balance() = balance()@pre and limit() = limit()@pre and balance() >= limit()) and not (balance()$_{pot}$ = balance()@pre – amount and limit()$_{pot}$ = limit@pre and balance()$_{pot}$ >= limit()$_{pot}$))

Figure 9 Generalizing the explicated specifications

2.3.4 Conclusion

We have learned that an invariant as such is an appropriate construct for modelling business rules that apply to objects of classes. The problem is that the semantics of an invariant do not match well with events. On the one hand, invariants cannot be ignored completely in the specification of events. On the other hand, complementing specifications of events with clauses related to invariants leads to duplication, which hampers adaptability and extendibility of conceptual models. Therefore we introduced the construct of a class constraint. An invariant and a class constraint must be both true at each moment in time. Additionally, the semantics of a class constraint is complemented with the semantics of the principle of non-violation. The concept of a class constraint leads to better adaptability and extendibility at the level of generalization as well as at the level of specialization. The approach with class constraints is less operational (more declarative) and less design-oriented than the approach with invariants and preconditions.

3 Modelling Restrictions on a Dynamic Properties

3.1 Problem Description

How can a restriction on a dynamic property be adequately specified? Consider the example in Figure 10. Assume that the amount involved in a withdrawal must always be at least 25 Euro. How do we specify this business rule?

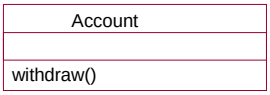


Figure 10 Modelling restrictions on dynamic properties

Four different approaches are discussed. The first approach uses preconditions, the second approach uses implications, and the third approach applies a combination of assertions in a postcondition to model a business rule for a particular event. The last approach introduces a new construct.

3.2 Anti-Patterns

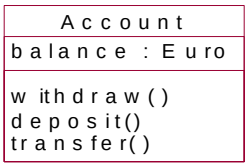
3.2.1 Preconditions

As demonstrated in the previous section, preconditions will not be used as a way to guarantee invariants. However, a precondition could still be used as a construct to impose a business rule on a particular event. In this subsection, the adequateness of preconditions for this purpose is investigated.

In the specification of the UML [OMG01, p.2-33], a precondition is defined as a constraint that must be true when an event occurs. Similarly, in Syntropy [Cook94, p.88] an event not meeting its preconditions just cannot happen. As an example, consider the definition of the event `withdraw()`, as it is specified in Figure 11. This event imposes a precondition on the amount to be withdrawn. This event is subsequently used in the compound event `transfer()` for transferring money from one account to another.

We want to evaluate the use of preconditions in combination with compound events, as we evaluated the use of invariants in combination with events. With the UML semantics for preconditions, compound events must ensure that the used events do not occur under conditions violating their preconditions. This may lead to situations in which compound events explicitly test preconditions. As for invariants, this results in a number of different places in the conceptual model where the same condition, the same

business rule is tested. As for invariants, this hampers the adaptability and the extendibility of conceptual model. Reconsider the event `transfer()` in Figure 11. Because this event is specified in terms of a withdrawal, the business-rule imposing a minimum on the amount involved in a withdrawal must be repeated at this point. This results in a precondition for the transfer-event. Obviously, if the limit must be increased to 30 Euro, or if additional business rules are imposed on the event `withdraw()`, the conceptual model must be changed at different places.



context Account :: withdraw(amount : Euro)
pre: amount >= 25 Euro
post: balance() = balance()@pre-amount

context Account :: deposit(amount : Euro)
post: balance() = balance()@pre + amount

context Account,Account :: transfer(amount : Euro)
pre: amount >= 25 Euro
post: self1.withdraw(amount)
self2.deposit(amount)

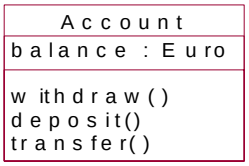
Figure 11 Preconditions

The definition of a precondition in the UML does not correspond with the more traditional definition [Dock99, p.114], which assumes that the state of the model is undetermined in case the precondition fails. Notice that differences in both definitions are subtle. In the first definition, it is forbidden to use the event if the precondition fails, while in the second it is allowed but the result is undetermined. In the additional literature about the UML [Rumb99, p.392] [Henn02, p.74], the definition of a precondition is inconsistent with the specifications in the UML and subscribes the more traditional view on preconditions.

The disadvantage in adopting the more traditional definition is that in case of a failure of the precondition the state of the model is indeterminate. This indeterminateness is too weak to be used for purposes of conceptual modelling. It leaves too much freedom. As an example, reconsider the definition of the event `withdraw()`. If the more traditional semantics apply to its precondition, the result of a withdrawal of an amount below 25 Euro is indeterminate. This means that the state of the model can be unchanged, but it can also be that the balance of the account is reset to the lowest possible value. This doesn't represent the observed real-world effects of the event.

3.2.2 Implications

In the second approach, an implication is used as a construct to impose a business rule on a particular event. In this subsection, the adequateness of an implication for this purpose is investigated. Due to the frame axiom, the state of the model remains unchanged if the condition of the implication fails. Consider the alternative specification of the event `withdraw()` as it is worked out in Figure 12. If the amount of money is smaller than 25 Euro the state of the model remains unchanged; if not, the balance of the account is decreased with the given amount.



context Account :: withdraw(amount : Euro)
post: amount >= 25 Euro implies balance()=balance()@pre-amount

context Account :: deposit(amount : Euro)
post: balance() = balance()@pre + amount

context Account,Account :: transfer(amount : Euro)
post: amount >= 25 Euro implies (self1.withdraw(amount) and self2.deposit(amount))

Explicated specification
context Account :: withdraw(amount : Euro)
e**post**: if (amount >= 25 Euro)
then balance() = balance()@pre – amount
else balance() = balance()@pre endif and

Account.AllInstances= Account.AllInstances@pre and --*frame axioms*
Account.AllInstances→excluding(self)→forAll(balance() = balance()@pre)

Figure 12 Implications

The approach using implications reveals three problems:

- As illustrated in the event `transfer()`, the use of implications cannot prevent that the same business rule is specified more than once. If the business rule is not repeated and the amount is smaller than 25 Euro the balance of one account will be increased with the given amount while the balance of the other account will be left untouched.

- Another disadvantage of this approach appears at the level of specialization. The condition of the implication cannot be strengthened at the level of a specialization without violating the principle of Liskov [Lisk94]. As illustrated in the explicated specification of the event `withdraw()`, the behaviour of this event cannot be further specialized without contradicting the assertions made at the level of generalization.

- Thirdly, the business rule and the effect of an event are mixed in one assertion. It is preferred to separate the specification of a business rule and the specification of the effect of an event in different assertions.

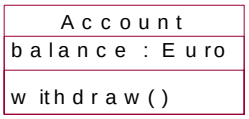
3.2.3 Postconditions

3.2.3.1 Postconditions representing Business Rules

A third alternative to model a business rule for a specific event is to model it as an assertion in a postcondition. In the UML, a postcondition is a constraint that must be true after the occurrence of an event, but the UML makes no statements about how to react if a postcondition is broken

[Warm99, p.6]. In the programming language Eiffel, an exception can be raised in case a postcondition does not hold [Meye97, p.413]. At the level of analysis, it can be assumed that in case some assertions of the postcondition do not hold, the state of the model is left unchanged.

Consider the example in Figure 13. Assume that the minimal amount for a withdrawal is 25 Euro. In the postcondition of the event `withdraw(),` both assertions cannot hold if a withdrawal with an amount smaller than 25 Euro occurs. Consequently, the balance of the account is left untouched in case of a withdrawal with an amount smaller than 25 Euro. An alternative for the second assertion of the postcondition can be found in comment. Notice that it is assumed that the amount itself cannot be changed.



context Account::withdraw(amount : Euro)
post: balance() = balance()@pre – amount
balance()@pre - balance >= 25 Euro -- *amount >= 25 Euro*

Figure 13 Postconditions

The problem with representing business rules by means of a postcondition is that the specification of the business rule and the specification of the effect of the event are mixed. This does not promote readability. An illustration of this readability problem is found in Figure 13. The business rule and the effect of the event cannot be easily distinguished from the combination of assertions in the postcondition. We are in favour of a clear difference between the effect of an event and a business rule imposed on an event.

3.2.3.2 Representing the Effect of an Event

In order to avoid that the specification of the effect of an event is mixed with the specification of business rules, the construct of an ***effect clause*** is introduced. All the assertions of an effect clause are guaranteed after the

occurrence of an event in case all (class) constraints hold. We do not consider the effect clause as a constraint; therefore the effect clause cannot be broken. Notice that during the design and implementation phase, postconditions are considered as constraints. During these phases, they can be broken. Failing postconditions can indicate implementation errors or exceptional circumstances.

Because the effect clause cannot be broken, it is invalid to specify assertions in the effect clause that cannot hold after the occurrence of an event. Consequently, assertions in an effect clause that may lead to a contradiction are not allowed. Consider the examples in Figure 14. In the first case, it is obvious that the assertions in the postcondition contradict each other and that the specification of the effect clause is invalid. In the second case, the assertion cannot hold after the occurrence of the event and is also invalid. In the third case, the assertions contradict each other when both accounts are identical and the amount to transfer is not zero. Notice that we do not read the assertions in a procedural way. If a transfer occurs from one account to the same account the specification states that the balance of that account must have been incremented and decremented at the same time with the given amount. In the last example, a business rule must be added indicating that the accounts involved in a transfer must be different.

Case one
context Class :: event()
effect: assertion() and not assertion()

Case Two
context Class :: event()
effect: false

Case Three
context Account :: withdraw(amount : Euro)
post: balance() = balance()@pre - amount

context Account :: deposit(amount : Euro)
post: balance() = balance()@pre + amount

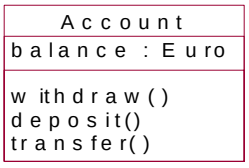
context Account,Account :: transfer(amount : Euro)
effect: self1.withdraw(amount)
self2.deposit(amount)

Figure 14 Effect clauses that cannot hold are rejected

3.3 Patterns

3.3.1 Event Constraints and the extended Principle of Non-Violation

We propose to model business rules that apply to a particular event, in terms of ***event constraints***. Consider the example in Figure 15, where the event constraint of the event `withdraw()` states that the minimum amount for a withdrawal is 25 Euro. In view of event constraints, the semantics of the principle of non-violation must be extended. The extended semantics states that an event, which is about to violate a constraint, does not change the state of the model. Consequently, an event does not only leave the state of the model unchanged when a class constraint is about to be violated but also when an event constraint is about to be violated. In case no constraints are about to be violated, the effect of the event is guaranteed. In our example, withdrawals with amounts smaller than 25 Euro are rejected and the state of the model does not change. In case the amount is not smaller than 25 Euro, the balance of the account will be decreased with the given amount (assuming that no other constraints are violated by the occurrence of the event).



context Account :: withdraw(amount : Euro)
const: amount >= 25 Euro
effect: balance()=balance@pre-amount

context Account :: deposit(amount : Euro)
effect: balance = balance@pre + amount

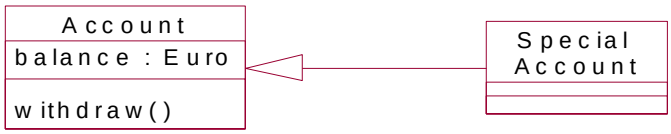
context Account,Account :: transfer(amount : Euro)
constr: self1 <> self2
effect: self1.withdraw(amount)
self2.deposit(amount)

Figure 15 Event constraints

The business rule `amount >= 25 Euro` must not be repeated for the event `transfer()`. If the amount of a transfer is smaller than 25 Euro an event constraint is violated. Consequently, the transfer is rejected and the state of the model is not modified. Obviously, extendibility and adaptability are promoted. Notice, that we added another event constraint for the event `transfer()`. As a result, the assertions in the postcondition no longer contradict each other.

3.3.2 Event Constraints at the Level of Specialization

An event constraint is a necessary condition for the effect of the event. In contrast with a precondition, which is a sufficient condition for the effect of the event. Necessary conditions are inherited conjunctively while sufficient conditions are inherited disjunctively [Wier91, p.415]. Consequently, event constraints are inherited conjunctively while preconditions are inherited disjunctively.



context Account :: withdraw(amount : Euro)
const: amount >= 25 Euro
effect: balance = balance@pre - amount

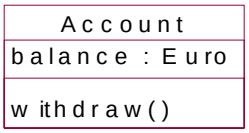
context Special Account :: withdraw(amount : Euro)
const: amount >= 50 Euro
effect: -- *is inherited*

Figure 16 Strengthening event constraints and implications

One of the advantages of a class constraint and the principle of non-violation is that one can easily strengthen or add class constraints at the level of the specialized class without modifying the specification of events defined at the level of a more general class. Event constraints have the same advantage. This advantage is illustrated in Figure 16. Event constraints can be strengthened without violating the principle of Liskov. At the level of special accounts an event constraint is stating that the minimum amount for a withdrawal is 50 Euro. At the level of accounts in general, the assertion states that the balance is only decreased if no constraints are violated. If a constraint is about to be violated, the state of the model is not changed. These semantics are not contradicted at the level of specialization.

3.3.3 Combining Class Constraints and Event Constraints

Event constraints and class constraints can be easily combined in a conceptual model. Because both kinds of constraints are necessary conditions for the effect of the event, the conditions for the effect of the event are combined using conjunction. This is illustrated in the explicated specification of the withdraw-event in Figure 17.



context Account **constraint**: balance() >= 1000 Euro

context Account: withdraw(amount: Euro)
const: amount >= 25 Euro
effect: balance() = balance()@pre - amount

Explicated Specification
context Account: withdraw(amount: Euro)
epost: if (balance@pre – amount >= 1000 Euro and amount >= 25 Euro)
then self.balance = self.balance@pre – amount
else self.balance = self.balance@pre endif and

Account.AllInstances→forAll(balance >= 1000 Euro) and --*class constraint*

Account.AllInstances= Account.AllInstances@pre and --*frame axiom*
Account.AllInstances→excluding(self)→forAll(balance = balance@pre)

Figure 17 Combining event constraints and class constraints

3.3.4 Generalizing the explicated Specification

In Figure 18, the explicated version of the effect of an event is given in general, taken into account the extended principle of non-violation and the frame axioms. The assertion `eventConstraints(event)` returns true if all involved event constraints for `event()` are satisfied. The involved event constraints of `event()` are all the explicitly stated event constraints of `event()` `itself`, all the inherited event constraints for `event()` and all involved event constraints of events that are used in the effect clause of the event `event()`. The assertion `succeed(event)` is an abbreviation for the assertion `(assertion() and frame()\event and classConstraints()` `and eventConstraints(event)@pre).` The assertion `fail(event)` is an abbreviation for the assertion $(\texttt{not(assertion()}_{pot}$ and $\texttt{frame()\event}_{pot}$ and $\texttt{classConstraints()}_{pot}$ and $\texttt{eventConstraints(event)}_{@pre})$.

In General

context Class :: event() -- *Shortcut version*
effect: assertion()

context Class :: event() -- *Explicated version (1)*
epost: (assertion() and frame()\event and classConstraints() and eventConstraints(event)$_{@pre}$) **or**
(frame() and (not (assertion()$_{pot}$ and frame()\event$_{pot}$ and classConstraints()$_{pot}$ and eventConstraints(event)$_{@pre}$))

context Class :: event() -- *Explicated version (2)*
epost: succeed(event) **or**
(frame() and fail(event))

An Example

context Account **constraint:** balance() >= limit()

context Account :: withdraw(amount: Euro) -- *Shortcut version*
eventc: not frozen()
effect: balance() = balance()@pre - amount

context Account :: withdraw(amount: Euro) -- *Explicated version*
epost: (balance() = balance()@pre – amount and limit() = limit()@ pre and frozen() = frozen@pre and balance() >= balance()@pre and not frozen()@pre) **or**

(balance() = balance()@pre and limit() = limit()@ pre and frozen() = frozen@pre and not ($balance()_{pot}$ = balance()@pre – amount and $limit()_{pot}$ = limit() and $frozen()_{pot}$ = frozen()@pre $balance()_{pot}$ >= $limit()_{pot}$ and not frozen()@pre))

Figure 18 Generalizing the explicated specification

3.3.5 Conclusions

During analysis, preconditions have no appropriate semantics or poses problems in terms of adaptability and extendibility. The use of an implication poses the same adaptability and extendibility problems as preconditions, and does not allow strengthening conditions at the level of specialization. If the assertions in the postcondition contradict each other, the contract principle is broken. Furthermore, the business rule and the effect of the event are mixed. Therefore we introduced the construct of an event constraint and extended the semantics of the principle of violation. As for class constraints, the concept of event constraints leads to better adaptability and extendibility at the level of generalization as well as at the level of specialization. The approach with event constraints is again more declarative, and less design-oriented than the approach with preconditions or implications.

Traditionally, preconditions, postconditions and invariants are considered as three concepts to model restrictions. We have argued that preconditions, postconditions and invariants should not be offered as constructs in object-oriented analysis. The use of these constructs during conceptual modelling may lead to adaptability, extendibility and consistency problems. Notice that no statements are made about the adequateness of preconditions, postconditions and invariants during the design phase.

In order to model business rules of all kinds, the general notion of a constraint has been introduced. Class constraints and event constraints are introduced as special kinds of constraints. Class constraints must be satisfied at all times by all the objects to which they apply; event constraints only apply to all occurrences of the event to which they apply. The principle of non-violation states that an event that is about to violate a constraint, will not cause any change to the objects involved. If no constraints are about to be violated, the effect of the event, which is specified in an effect clause, is guaranteed. We have further argued that an effect clause must only be used to describe the effect of an event, and may not be used to describe business rules. An effect clause describes the effect of the occurrence of an event in case all constraints are satisfied. The effect clause must be written in a way that no contradictions can occur. In such a context, effect clauses are not considered as constraints.

Compared with the traditional approach, the approach with constraints specifies restrictions at only one place. As a result, the adaptability and extendibility of a conceptual model is increased. Furthermore, this approach allows specializing class constraints and event constraints without the need of changing existing specification of the generalized classes. The constraints are considered as necessary conditions and are inherited and combined conjunctively.

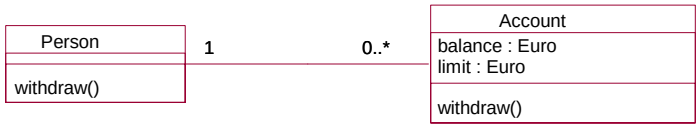
During design, a conceptual model must be transformed in a computational model. This transformation must be based on software quality factors and non-functional requirements. At the level of design, preconditions, postconditions and invariants can be used as constructs to transform the constraints of the conceptual model. In such a case, a computational model reveals adaptability and extendibility problems. However the aim of this pattern was to resolve these problems at the level of analysis. At the level of implementation, a software engineer must decide how to realize the preconditions, postconditions and invariants that are present in the computational model. Defensive programming, total programming and nominal programming are possible implementation strategies at that stage.

4 Relaxation of the Principle of Non-Violation

4.1 Problem Description

The principle of non-violation states that an event that is about to violate a constraint is rejected. Consequently, if *at least one* of the assertions of the effect clause of an event cannot be satisfied, the event will be rejected in its entirety. In other words, events have ***transactional semantics*** [Elma00, p. 629-659]. The notion of a transaction does not exist in the UML [OMG01] but other authors, for instance [Ngu89], introduced this notion in conceptual modelling. However, it is still rarely used in conceptual modelling.

Consider an example in Figure 19. A person wants to withdraw a certain amount of money from all his accounts. If at least one of his accounts has an insufficient balance, all his accounts remain untouched due to the violation of the *class constraint*. This could be the desired semantics of the event `withdraw()` attached to the class `Person`. But assume the desired semantics are to withdraw a certain amount from all accounts that permit such a transaction. How can such an event be specified?



context Account **constraint:** balance() >= limit()

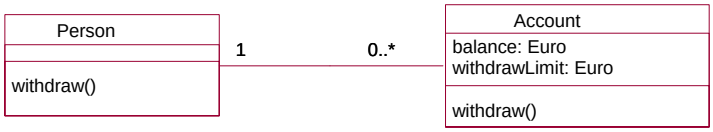
context Account :: withdraw(amount : Euro)
effect: balance() = balance()@pre - amount

context Person :: withdraw(amount : Euro)
effect: account().withdraw(amount)

Figure 19 Transactional semantics for events (1)

Consider another example in Figure 20. A person wants to withdraw a certain amount of money from all his accounts. If at least one of his accounts has an insufficient balance, all his accounts remain untouched due to the violation of the *event constraint*. This could be the desired semantics of the

event `withdraw()` attached to the class `Person`. But assume the desired semantics are to withdraw a certain amount from all accounts that permit such a transaction. How can such an event be specified?



context Account :: withdraw(amount: Euro)
const: amount <= withdrawLimit()
effect: balance() = balance()@pre – amount

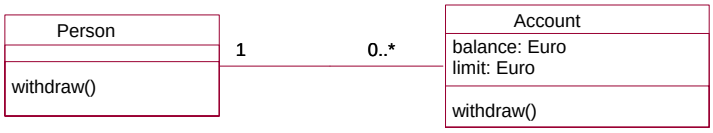
context Person :: withdraw(amount: Euro)
effect: account().withdraw(amount)

Figure 20 Transactional semantics for events (2)

4.2 Anti-Patterns

4.2.1 Selecting the appropriate Set

A possible approach to model such an event to withdraw some amount of money from all accounts with a sufficiently high balance is to select carefully the concerned accounts. This approach is adopted in Figure 21. In the specification of the event at the level of the class `Person`, all accounts of a person with a sufficient balance are selected first. The mutator `withdraw()` of the class `Account` is then applied on this carefully selected set of accounts.



context Account **constraint:** balance() >= limit()

context Account :: withdraw(amount : Euro)

effect: balance() = balance()@pre - amount

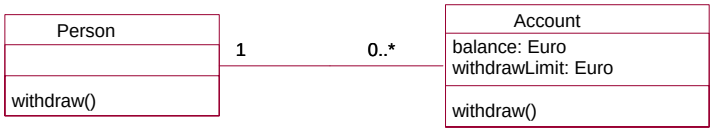
context Person :: withdraw(amount : Euro)
effect: account()→select(balance()–amount >= limit()).withdraw(amount)

Figure 21 Selecting the appropriate set

This approach does not support extendibility and adaptability. If new constraints are added or existing constraints are changed, the selection must be changed as well in order to preserve the desired semantics. Once again, a business rule is modelled at several places.

4.2.2 Using the Implication

On could argue that transactional semantics for events is an argument to allow the use of an implication. A compound event has different semantics as it uses an event with an implication or as it uses an event with an event constraint. Consider the example in Figure 22. The event `withdraw()` attached to the class `Person` has different semantics compared with the same event in Figure 20. In Figure 20, the event will leave the state of the model unchanged if at least one of the accounts of the given person has an insufficient balance. In Figure 22, the event will withdraw the given amount from all accounts for which the balance is sufficiently high and leaves the state of the model only unchanged if all of the accounts of a given person have an insufficient balance.



context Account :: withdraw(amount: Euro)
effect: amount <= withdrawLimit() implies balance() = balance()@pre - amount

context Person :: withdraw(amount: Euro)
effect: account().withdraw(amount)

Figure 22 Using the implication

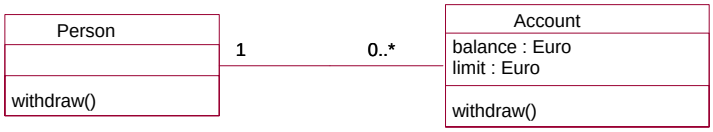
Aside the disadvantages mentioned in section 3.2.2 with the use of the implication, another disadvantage is that two events `withdraw()` could be attached to the class Account: an event `withdraw()` using an implication and an event `withdraw()` using an event constraint. To conclude, the use of the implication is discouraged in our approach. Notice that instead of using an implication, the analyst could also have selected the appropriate subset of accounts.

4.3 Patterns

4.3.1 Events with non-transactional Semantics

In order to support extendibility and adaptability and to avoid the use of implications, the construct of ***non-transactional semantics*** for events is introduced. The principle of non-violation is relaxed for events with non-transactional semantics. If non-transactional semantics for an event is used, the principle of non-violation states that an event in its entirety is only rejected if *all* assertions violate at least one constraint.

Consider the examples in Figure 23 and in Figure 24. An event `withdraw()` with non-transactional semantics is specified in the class `Person`. Assertions with non-transactional semantics are combined with the explicit symbol ⊕. In the examples, the effect clause of the compound event `withdraw()` is denoted as `account()⊕ withdraw()`. This event leaves the state of the model unchanged if *all* withdrawals on the accounts of a person are rejected. Furthermore, one is sure that the balance of a persons account will be decreased with the given amount if this withdrawal does not violate a constraint.

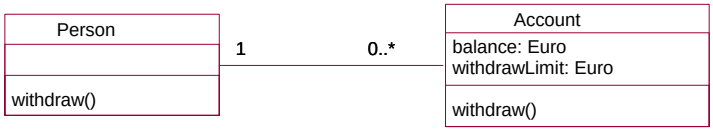


context Account **constraint:** balance() >= limit()

context Account :: withdraw(amount : Euro)
effect: balance() = balance()@pre – amount

context Person :: withdraw(amount : Euro)
effect: account()⊕withdraw(amount)

Figure 23 Non-transactional semantics for events (1)



context Account :: withdraw(amount: Euro)
const: amount <= withdrawLimit()
effect: balance() = balance()@pre – amount

context Person :: withdraw(amount: Euro) -- *non-transactional*
effect: account⊕withdraw(amount)

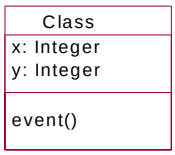
Figure 24 Non-transactional semantics for events (2)

Notice that the event `withdraw()` attached to the class `Person` in Figure 19, in Figure 21 and in Figure 23 have different *semantics* in each of the three cases. In Figure 19, the analyst wants to model an event that withdraws money from all the accounts of a person. In Figure 21, an event is modelled that withdraws money from a subset of accounts that have a sufficient balance. In Figure 23, money is withdrawn from all accounts where it is possible to do so. This matches with reality where a person can state: "I want to withdraw money from all my accounts" versus "I want to withdraw money from all accounts with a sufficient balance" versus "I want to withdraw money from all accounts where it is possible to so". The introduction of transactional and non-transactional semantics for an event helps to remove possible ambiguity when natural language is used. In our example, what is the precise meaning if somebody talks about "*all* accounts"?

4.3.2 Non-determinism

The use of the non-transactional operator ⊕ can lead to ***non-determinism***. Non-determinism can occur in case a certain subset of assertions of the event exists, in which each assertion on its own satisfies all constraints, but where the subset of assertions as a whole violates at least one constraint. In such a case, only a certain subset of this subset of assertions is satisfied. This subset of subset of assertions must satisfy all constraints.

Consider the example in Figure 25. A class constraint `x() >= y()` and an event decreasing `x` and increasing `y` is present. Three different cases are illustrated in the comments. In the second case, each assertion of the event can in isolation occur without violating a constraint but both assertions cannot be satisfied together. This leads to a non-determined situation where only one of both assertions will be satisfied. The first case and third case are determined. In the first case, both assertions are satisfied. In the third case, no assertions are satisfied.



context Class **constraint:** x() >= y()

context Class event(a: Integer)
effect: x() = x()@pre – a ⊕ y() = y()@pre + a

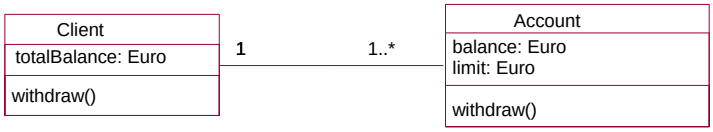
-- case 1
-- x()@pre = 200 and y()@pre = 100 and a = 20
-- x() = 180 and y() = 120

-- case 2: non-determinism
-- x()@pre = 200 and y()@pre = 100 and a = 70
-- (x() = 130 and y() = 100) ***or*** *(x()= 200 and y() = 170)*

-- case 3
-- x()@pre = 200 and y()@pre = 100 and a = 150
-- x() = 200 and y() = 100

Figure 25 Non-determininsm with the non-transactional operator ⊕

Furthermore, as much assertions as possible of the subset of assertions, in which the assertions in isolation satisfy the constraints, must be satisfied. Consider the example in Figure 26. Assume that a certain client has ten accounts and that his total balance equals 2.000 Euro. Six of the ten accounts allow withdrawing 1000 Euro in isolation without violating the constraints. Obviously, if all these withdrawals occur the total balance of the client equals – 4000 Euro and violates the class constraint about total balances. In such a case, withdrawals on only four of the six accounts will take place.



context Account **constraint:** balance() >= limit()
context Client **constraint:** totalBalance() >= - 2500 Euro
context Client **constraint:** totalBalance() = account().balance()→sum()

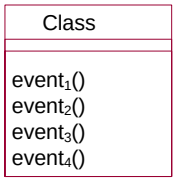
context Account :: withdraw(amount : Euro)
effect: balance() = balance()@pre - amount

context Client :: withdraw(amount : Euro)
effect: account()⊕withdraw(amount)

Figure 26 As much assertions as possible must be satisfied

4.3.3 Combining Transactional with Non-Transactional Semantics

Transactional semantics, denoted by the implicit and-operator and non-transactional semantics, denoted by the ⊕-operator, can be combined in an event. The precedence order for the '⊕' is the same as for the conjunction. Parentheses can be used to change precedence. This could lead to very complex specifications. It is unlikely that the analyst will often need such combinations. Consider the events in Figure 27. For each event the different possible cases are given in the comments. Notice that the involved event constraints for assertion $ass_1()$ in event $event_3()$ are the event constraints $eventconstraint_1()$ and $eventconstraint_3()$ and not $eventconstraint_2()$.



context Class **constraint:** clasconstraint()

context Class :: $event_1()$
const: $eventconstraint_1()$
effect: $ass_1()$ $ass_2()$

context Class :: $event_2()$
const: $eventconstraint_2()$
effect: $ass_3() \oplus ass_4()$

context Class :: $event_3()$
const: $eventconstraint_3()$
effect: $event_1()$ $event_2()$ *--($ass_1()$ and $ass_2()$) and ($ass_3() \oplus ass_4()$)*

-- case 1: $event_1$ and $event_2$ succeed => $event_3$ succeeds
-- all assertions are satisfied

-- case2: $event_1$ and $event_2$ succeed => $event_3$ succeeds
-- all assertions are satisfied except ass_3 fails

-- case3: $event_1$ and $event_2$ succeed => $event_3$ succeeds
-- all assertions are satisfied except ass_4 fails

-- case4: $event_1$ or $event_2$ are rejected => $event_3$ is rejected => frame()
-- (ass_1 or ass_2) or (ass_3 and ass_4) fail

context Class :: $event_4()$
const: $eventconstraint_4()$
post: $event_1() \oplus event_2()$ *--($ass_1()$ and $ass_2()$) $\oplus$ ($ass_3() \oplus ass_4()$)*

-- case1: $event_1$ and $event_2$ succeed => event4 succeeds
-- all assertions are sastisfied

-- case 2: $event_1$ and $event_2$ succeed => $event_4$ succeeds
-- all assertions are satisfied except ass_3 fails

-- case 3: $event_1$ and $event_2$ succeed => $event_4$ succeeds
-- all assertions are satisfied except ass_4 fails

-- case 4: $event_1$ succeeds and $event_2$ is rejected => $event_4$ succeeds
-- (ass_1 and ass_2) are satisfied and (ass_3 and ass_4) fail

-- case5: $event_1$ is rejected and $event_2$ succeeds => $event_4$ succeeds
-- (ass_1 or ass_2) fail and (ass_3 and ass_4)are satisfied

-- case6: $event_1$ is rejected and $event_2$ succeeds => $event_4$ succeeds
-- (ass_1 or ass_2) fail and ass_3 is satisfied and ass_4 fails

-- case7: $event_1$ is rejected and $event_2$ succeeds => $event_4$ succeeds
-- (ass_1 or ass_2) fail and ass_3 fails and ass_4 is satisfied

-- case8: $event_1$ and $event_2$ are rejected => $event_4$ is rejected => frame()
-- (ass_1 or ass_2) and (ass_3 and ass_4) fail

Figure 27 Combining transactional and non-transactional semantics

5 Relaxation of the Frame Axiom

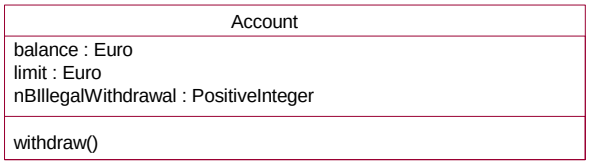
5.1 Problem Description

Until now, it was assumed that in case of a rejection of an event the state of the model was left unchanged. This assumption is sometimes invalid. How can a change in the state of the model be specified in case an event is about to violate a constraint?

5.2 Anti-Patterns

5.2.1 If-then-else Statements

An if-then-else statement in the effect clause of an event can be used to specify a change in the state of the model in case an event is about to violate a constraint. Consider the example in Figure 28. Assume that the number of times a withdrawal is rejected due to a constraint is a relevant. In the example, the characteristic `nBIllegalWithdrawal` is augmented with one in the else statement of the effect clause of the event `withdraw()`.



context Account **constraint:** balance() >= limit()

context Account :: withdraw(amount : Euro)
effect: if balance()@pre – amount >= limit
then balance() = balance()@pre - amount
else nBIllegalWithdrawal() = nBIllegalWithdrawal()@pre + 1
endif

Figure 28 Using if-then-else statements

Once again, adaptability and extendibility are not supported. If new constraints must be added or existing constraints must be changed, the condition of the if-then-else statement must also be changed.

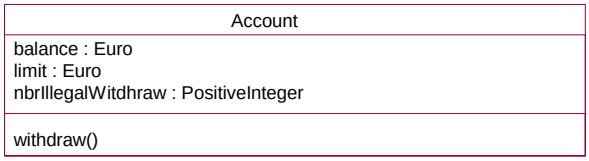
5.3 Patterns

5.3.1 A Rejection Clause for an Event

In our approach, it is possible to specify what happens in case an event is rejected by no matter which constraint. Aside an effect clause and an event constraint, a ***rejection clause*** can be specified for an event. The default situation is that the state of the model is unchanged. In this case, the rejection clause is normally left out. At the moment, we assume that such rejections clauses are not contradicted by class constraints. More research is needed to establish the semantics of rejection clauses that can violate other constraints.

Consider the example in Figure 29. The number of illegal withdrawals is increased each time a withdrawal is rejected. Notice that the approach with the if-then-else statement and the approach with the rejection clause have different semantics. In the approach with the rejection clause, the number of illegal withdrawals is increased in case a withdrawal is rejected if it is about

to violate at least one business rule. In the approach with the if-then-else statement, the number of illegal withdrawals is only increased in case of an insufficient balance.



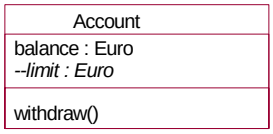
context Account **constraint:** balance() >= limit()

context Account :: withdraw(amount : Euro)
effect: balance() = balance()@pre – amount
reject: nbrIllegalWithdraw() = nbrIllegalWithdraw()@pre + 1

Figure 29 Example of a rejection clause for an event

5.3.2 Misusing Rejection Clauses for Events

One could argue that if-then-else statements are no longer useful and that rejection clause and event constraints can replace if-then-else statements in specifications. This is incorrect. Consider the example in Figure 30. Assume that for a withdrawal of an amount of at least 1000 Euro a cost of 0.2 Euro is taken into account. In the first alternative, an if-then-else statement is used to specify the event `withdraw()`. In the second and third alternative, an event constraint in combination with a rejection clause is used to specify this event. One could argue that the second and third alternative support extendibility because event constraints can be strengthened at the level of specialization.



-- context Account constraint: balance() >= limit()

```
-- Alternative One
context   Account :: withdraw(amount : Euro)
effect:   if      (amount <= 1000 Euro)
          then    balance() = balance()@pre – amount
          else    balance() = balance()@pre – amount – 0.2 Euro
          endif

-- Alternative two: misusing rejection clauses
context   Account :: withdraw(amount: Euro)
const:    amount <= 1000 Euro
effect:   balance() = balance()@pre – amount
reject:   balance() = balance()@pre – amount – 0.2 Euro

-- Alternative three: misusing rejection clauses
context   Account :: withdraw(amount : Euro)
const:    amount > 1000 Euro
effect:   balance() = balance()@pre – amount – 0.2 Euro
reject:   balance() = balance()@pre – amount
```

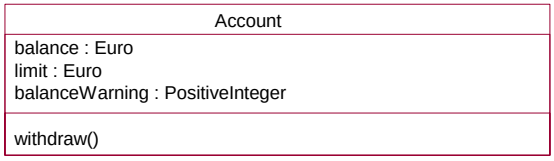
Figure 30 Misusing rejection clauses

The last two alternatives in Figure 30 are invalid. In case new constraints are added to the model, the rejection clause of the event occurs in case the event is rejected due to the newly added constraints. This would not be the desired semantics. An example of a possible added class constraint is given in comment. It is not because rejection clauses may not be misused that we are in favour of the use of if-then-else statements. We are investigating new patterns that allow avoiding if-then-else statements.

5.3.3 A Rejection Clause for a Class Constraint

It should be also possible to specify what happens in case a class constraint is about to be violated by no matter which event. Therefore a rejection clause for a class constraint is introduced. Aside an assertion that must be satisfied at all times, a rejection clause can be specified for a class constraint. If a class constraint is about to be violated, the default situation is that the state of the model is unchanged. In this case, the rejection clause is normally left out.

Consider the example in Figure 31. The characteristic `balanceWarning` of an account represents the number of times the business rule "balance exceeds limit" is about to be violated by no matter which event. Notice that an if-then-else statement in the effect clause of the event `withdraw()` is not a good modelling option for extendibility and adaptability reasons. Indeed, other events may exist or may be introduced later on, that could also lead to a violation of the constraint concerning the balance of an account.



context Account **constraint:** balance() >= limit()
reject: balanceWarning() = balanceWarning()@pre + 1

context Account :: withdraw(amount : Euro)
effect: balance() = balance()@pre - amount

Figure 31 An example of a rejection clause for a class constraint

If an event is about to violate more than one class constraint with a rejection clause, then all assertions in the rejection clauses are conjunctively combined. Rejection clauses are also conjunctively combined in case of an event with a rejection clause is about to violate a class constraint with a rejection clause.

5.3.4 Rejection Clauses and (non-) transactional Semantics

Rejection clauses can be combined with transactional or non-transactional semantics of events. Consider event $event_3()$ in Figure 32. The semantics of an event with transactional semantics and a rejection clause consists of combinations of its own rejection clause and of the rejections clauses of the composing events in case the overall event is rejected. Consider event $event_4()$. The semantics of an event with non-transactional semantics and a rejection clause consists of combinations of assertions in the effect clause of composing events that succeed and of assertions in the rejection clause of

composing events that are rejected. Only in the case that all composing events fail, the rejection clause of the overall event with non-transaction semantics also applies.

context Class **constraint:** classconstraint()

context Class :: $event_1()$
const: $eventconstraint_1()$
effect: $ass_1()$
reject: $rejection_1()$

context Class :: $event_2()$
const: $eventconstraint_2()$
effect: $ass_2()$
reject: $rejection_2()$

context Class :: $event_3()$
const: $eventconstraint_3()$
effect: $event_1()$ $event_2()$
reject: $rejection_3()$

-- case1: $event_1$ and $event_2$ succeed => $event_3$ succeeds
-- ass_1 and ass_2 are satisfied

-- case 2: $event_1$ succeeds and $event_2$ is rejected => $event_3$ is rejected
-- ass_2 fails and $rejection_2$ and $rejection_3$ are satisfied

-- case 3: $event_1$ is rejected and $event_2$ succeeds => $event_3$ is rejected
-- ass_1 fails and $rejection_1$ and $rejection_3$ are satisfied

-- case 4: $event_1$ and $event_2$ are rejected => $event_3$ is rejected
-- ass_1 and ass_2 fail and $rejection_1$ and $rejection_2$ and $rejection_3$ are satisfied

context Class :: $event_4()$
const: $eventconstraint_4()$
effect: $event_1() \oplus event_2()$
reject: $rejection_4()$

-- case1: $event_1$ and $event_2$ succeed => $event_3$ succeeds
-- ass_1 and ass_2 are satisfied

-- $event_1$ succeeds and $event_2$ fails => $event_3$ succeeds
-- ass_1 is satisfied ass_2 fails and $rejection_2$ is satisfied

-- $event_1$ fails and $event_2$ succeeds => $event_3$ succeeds
-- ass_1 fails and $rejection_1$ is satified and ass_2 is satisfied

-- $event_1$ and $event_2$ are rejected => $event_3$ is rejected
--ass_1 and ass_2 fail and $rejection_1$ and $rejection_2$ and $rejection_3$ are satisfied

Figure 32 Rejection clauses and (non-) transactional semantics

6 Implicitly or Explicitly modelled Class Constraints

6.1 Problem Description

Some class constraints can be modelled explicitly or implicitly. Explicit class constraints are textually modelled with the help of the OCL. Implicit class constraints are visually modelled. Notice that the principle of non-violation also applies to implicitly modelled class constraints.

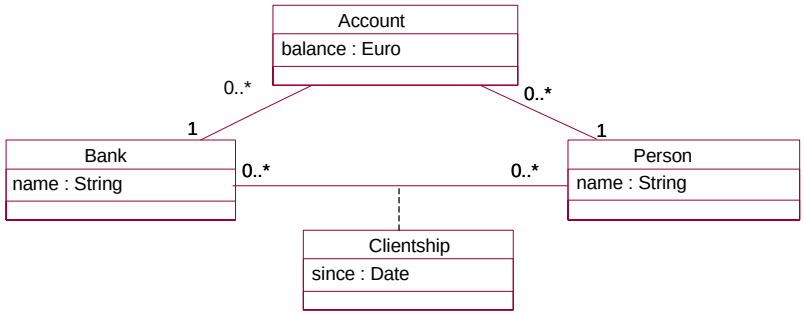
Warmer and Kleppe [Warm99, p.73-75] make no choice between implicitly or explicitly modelled class constraints. They only mention a trade-off between "adding details to the class diagram" and "adding textual constraints". In EROOS [Steeg00, p.121], explicit class constraints must be avoided whenever possible. Other authors [Kent99][Bott01] introduce new constructs to model constraints as visual as possible. Do we model class constraints always implicitly, always explicitly or does the preferred solution depend on the situation?

6.2 Anti-Patterns

6.2.1 Explicitly modelled Class Constraints.

Consider a first example in Figure 33. Persons can have accounts at a bank.

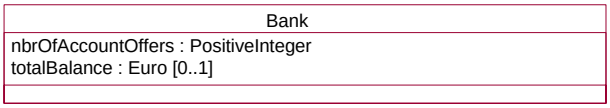
Furthermore, a clientship relation can exist between a bank and a person. The restriction that a person cannot have accounts at a bank without being a client is explicitly modelled in the class constraint.



context Account **constraint:**
bank().clientship(),person().clientship()→intersection()→notEmpty()

Figure 33 Explicitly modelled class constraints (1)

Consider a second example in Figure 34. A bank has as properties the number of account offers and a total balance. Between both properties a dependency exist, which is specified in the explicit class constraint.



context Bank **constraint:** nbrOfAccountOffers() = 0 implies totalBalance().empty()

Figure 34 Explicitly modelled class constraints (2)

We are not in favour of modelling constraints explicitly. The evaluation between explicitly and implicitly modelled class constraints is made in section 6.3. In chapter two, it was already stated that class constraints testing the number of elements must be avoided.

6.3 Patterns

6.3.1 Case one: all Model Elements are relevant

If all model elements in a conceptual model are relevant, implicit class constraints are preferred. This is already described by Van Baelen et al. [Vanb94, p.190]. They argued that the basic structure of the conceptual model, one of the important results of analysis, is neglected and shifted to a combination of implicit constraints and explicit constraints if explicit class constraints are not avoided as much as possible.

Compare the conceptual model in Figure 35 with the model in Figure 33. In Figure 35, the restriction that one cannot have accounts without being a client is implicitly modelled. This is less complex and easier to interpret compared with the explicitly modelled constraint in Figure 33. Notice that for the same reason an association class `Clientship` is modelled instead of a combination of a class `Clientship` with an explicit class constraint restricting the number of client ships between a person and a bank. In [Gogo98, p.95] such kind of transformation is described, but no choice between implicitly or explicitly modelled class constraints is made.

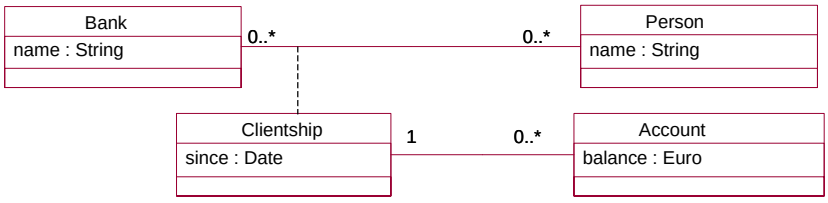
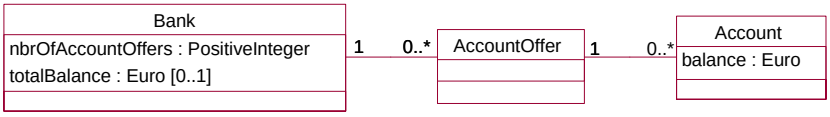


Figure 35 Implicitly modelled class constraints

6.3.2 Second case: not all Model Elements are relevant

If not all model elements are relevant, implicit constraints are still preferred. Consider the example in Figure 36. The conceptual model is an extension of the model of in Figure 34. The classes `AccountOffer` and `Account` are

added. Assume that the set of account offers and the set of accounts are not relevant. These classes were introduced to model implicitly the explicit class constraint in Figure 34. One could argue that in Figure 36 the principle of abstraction is violated. This is correct. On the other hand, if the model in Figure 33 would be allowed, the principle of uniqueness or the principle of extendibility would be violated in case the model is extended later on with the classes `Account` and `AccountOffer`. If the constraint `nbrOfAccountOffers() = 0 implies totalBalance().empty()` of the second model in Figure 33 is left untouched in case of the extension, the principle of uniqueness is violated because the restriction is two times represented. If the constraint is deleted in case of the extension, the principle of extendibility is violated because existing specifications are modified or deleted.



context Bank **constraint:** totalBalance() = account().balance()→sum()
context Bank **constraint:** nbrOfAccountOffers() = accountOffer()→size()

Figure 36 Implicitly modelled class constraints (2)

Two patterns violate the principle of abstraction. The first pattern recommended attaching model elements to classes instead of to data types. The second pattern recommends modelling implicit constraints instead of explicit constraints. In both patterns, the principle of abstraction is violated because implicit constraints and attaching model elements to classes generally lead to better-structured models. In our example, we do not want a conceptual model containing only the class `Bank` having a lot of properties involving account offers or involving accounts. This would lead to a lot of dependencies between these properties expressed by means of explicitly modelled class constraints.

One could argue that in Figure 36, there are still two class constraints present. This is correct, but these class constraints are modelling information

redundancy relationships and cannot be avoided if information redundancy should be supported.

Chapter 6

Patterns for Modelling Classification

Classification is defined as the grouping of objects into classes, or more generally, as the grouping of instances into types. The fundamental construct of a ***specialization-generalization relationship*** in object orientation allows an analyst to classify an object into several classes, reflecting that objects of a specialized class are *special kinds of* objects of a more general class. In this chapter, this construct will be the focus of interest.

A specialization-generalization relationship between classes can be specified. The term ***superclass*** is used for the more general class, the term ***subclass*** for the more specific class [Rumb99, p.287]. The subclass has all the model elements attached to the superclass and may introduce additional model elements. Moreover, this relationship offers better support for reusability and extendibility. Generalization-specialization allows the analyst to extend its conceptual model without modifications of what has already been modelled, and at the same time to reuse elements of an existing conceptual model.

Although not further discussed in this chapter, we believe that both ***dynamic classification*** and ***multiple classification*** must be supported at the level of analysis. Static classification implies that an object may not leave its class after its creation. We fully agree with [Rumb99, p.54], where it is argued that there is no logical necessity for this restriction, and that dynamic classification is a valuable modelling construct although not supported by many programming languages.

Multiple classification allows a subclass to have more than one direct superclass. Similar remarks as those for dynamic classification can be made for multiple classification. In [Beka02, p.34], we stated that multiple specialization-generalization is only a logical extension of single specialization-generalization and not an optional or exotic construct.

Although many programming languages do not support multiple classification considering it as a rather problematic concept, this construct is essential during conceptual modelling. It allows representing real-world objects from many viewpoints.

In section 1, the semantics of the specialization-generalization relationship and its consequences are discussed. In section 2, the question is tackled when to generalize classes and/or model elements of classes. Model elements of a class can be characteristics, associations, events or class constraints. Section 3 elaborates how to model particulars in a conceptual model.

1 The Generalization-Specialization Relationship

1.1 Problem Description

The generalization-specialization relationship is one of the key constructs in object-orientation. Generalization and specialization are each mirror image. From this point on, the generalization viewpoint will be taken. Unfortunately, the semantics of this relationship is not defined with the same accuracy [Duco02, p.3].

In the UML [OMG01, p.2-39], generalization is defined as a taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and may contain additional information. It is argued that such a taxonomic relationship can be detected via classes that have common model elements or/and via plain common sense [Steeg00, p.128]. Furthermore, generalization is a ***subtype relationship***; that is, an object of the superclass may be substituted by an object of the subclass.

The definition of generalization is too vague and can lead to misuse of the generalization construct. Examples of such misuses are given in the subsection 1.2. In subsection 1.3, we try to be more accurate concerning the precise semantics of a generalization relationship. At that time, we will study the consequences of generalization for the model elements defined at the level of a superclass and at the level of a subclass.

1.2 Anti-Patterns

1.2.1 Common Model Elements

As stated in [Rumb99, p.52] and in [Stee00, p.128], a basic purpose of generalization is inheritance. Inheritance allows representing common model elements only once at the level of the superclass and thus avoiding redundancy. If this purpose is emphasized, a generalization relationship such as the one illustrated in Figure 1 could be envisaged. In the model, a generalization relationship is drawn as an arrow from the subclass to the superclass. The subclass `Person` inherits the model elements `name` and `changeName`. These model elements are only declared once at the level of the superclass `Bank`.

Figure 1 Common model elements

This kind of generalization would lead to "spaghetti-generalization". If a new characteristic of a bank such as a rating must be added to the conceptual model, the analyst would be forced to contradict the inheritance of this characteristic in the class `Person`. Moreover, the direction of the generalization would become arbitrarily. The class `Bank` could also be a subclass of the class `Person` and the inheritance of the characteristic `weight` would be contradicted in the class `Bank`.

This anti-pattern is inline with Meyers "is-a rule" [Meye97, p.811]. In this rule, Meyer advises not to define a generalization between a superclass `A` and a subclass `B` unless one can somehow make the argument that every object of the subclass `B` can also be viewed as an object of the superclass `A`. In the example of Figure 1, one cannot consider a person to be a bank, nor vice versa.

1.2.2 Adding Model Elements

An often-occurring situation is that of a model that must be extended. Consider the example in Figure 2. Assume that the class `Account` and the characteristic `balance` are already modelled. After some time, the analyst wants to represent that *each* account also has a limit. With the construct of generalization, an analyst could introduce a new subclass. If the subclass represents the same set of objects as the superclass, then the new model element can be attached to the subclass. In the example, a new subclass `Account+` with a characteristic `limit` is added to the model. The constraint `{complete}` [OMG01, p.41] indicates that each object of the superclass `Account` is also an object of the subclass `Account+`.

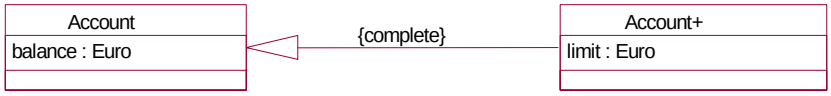


Figure 2 Adding model elements

In case a subclass and a superclass represent the same set of objects, the same observed real-world fact, namely the set of objects, is represented more than once. The subclass `Account+` is only a technical construct to attach the characteristic `limit` to it and has no semantic foundation. Furthermore, the direction of the arrow of the generalization relationship is semantically arbitrarily. We believe that generalization is not an adequate construct to represent such extensions. In programming this may be different, in particular in case the source code is not available. A consequence of this anti-pattern is that a *complete* subdivision must have at least two subclasses, representing different sets of objects.

1.3 Patterns

As stated in [Beka02, p.35], the primary characteristic of a generalization relationship between two classes is that it introduces a permanent set

inclusion relationship between the subclass and the superclass. The set inclusion has two important logic consequences. These consequences are described in the next paragraphs:

- The ***"inheritance-consequence"***: all model elements of the superclass are inherited by each of its subclasses. If a model element of the superclass would not be inherited by a subclass, then the set of objects of the subclass would no longer be a subset or that set would always be empty. A class, which represents a set of objects that is always empty, is by definition not relevant.

- The ***"refinement-consequence"***: the specification of the inherited model elements can only be refined and not contradicted at the level of the subclass. If a specification at the level of the subclass would contradict the specification at the level of the superclass, then the set of objects of the subclass would again no longer be a subset or the set would always be empty.

Notice, that no statement is made here about how to refine inherited model elements. Several approaches for refining inherited model elements exist. As an example, a methodology can offer the possibility to represent exceptions explicitly [Borg88], a methodology can offer preconditions, that can be weakened at the level of subclasses, or a methodology can offer event constraints that can be strengthened at the level of subclasses.

An immediate consequence of the subset relationship is that it enables polymorphism. During analysis, we define polymorphism as the capability to reason about a superset of objects disregarding the actual kinds of objects in that superset. Polymorphism also implies the capability to refine such reasoning at the level of subclasses. If contradictions at the level of a specialization would be allowed or if it would be possible not to inherit some model element of a superclass at the level of a specialization, one looses the subset inclusion and therefore also, polymorphism.

Consider the example outlined in Figure 3. The set of current accounts is a subset of the set of accounts. Consequently, if a statement is made about accounts in general, it is also made about all current accounts. Furthermore,

no statements about current accounts can be made that contradict statements about accounts. In the comments, the generalization relationship is explicitly modelled as a class constraint. Recall that a class constraint must *always* be satisfied. In the example, the set of current accounts must always be a subset of the set of accounts. Throughout the rest of this thesis, a generalization relationship is represented by an arrow. That arrow implies among others the class constraint illustrated in Figure 4.

*-- **context** CurrentAccount **constraint**: Account.allInstances()→includes(self)*

Figure 3 Generalization

Notice that not all set inclusion relationships are worked by means of generalization. A set of objects of a class can be temporarily a subset of the set of objects of another class. Consider an example in which all of my friends are currently football players. The class `Myfriend` cannot be specified as a subclass of the class `Footballer`. Otherwise, it would be impossible to make new friends who are not footballers, and it would be impossible for my friends to stop playing football without being my friend.

Because of the important consequences of the subset relationship enabling polymorphism, it is fundamental to be able to represent this subset relationship in conceptual modelling. Object orientation supports the representation of this relationship via generalization. We are convinced that this is one of the major reasons why object orientation has become hegemonic in the field of software engineering [Duco02, p.3].

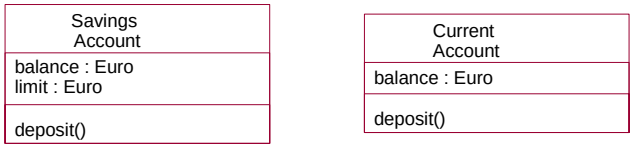
2 To generalize or not to Generalize Classes and Model Elements of Classes

2.1 Problem description

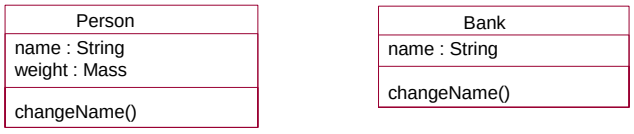
In section 1, the semantics of a generalization relationship have been defined and its consequences are explained. Then the question arises when to generalize classes and/or model elements of classes? In [Rumb99, p.51-52], it is argued that generalization has two purposes. A first purpose of generalization is the need for polymorphism. As defined in section 1, polymorphism during the analysis phase is the capability to reason about a superset of objects and the capability to refine this reasoning at the level of subclasses. The second purpose of generalization is inheritance. Inheritance allows incorporating all the model elements of a superclass in subclasses without the need to introduce these model elements in an explicit way at the level of these subclasses.

Consider the cases outlined in Figure 4. If the analyst focuses on the purpose of inheritance, then the class `SavingsAccount` and `CurrentAccount` should be generalized into a class `Account`. At the same time, the class `Person` and `Bank` can be generalized into a class `BankPerson`, representing the set of persons and banks. The class `House` and `Car` cannot be generalized because there are no relevant common model elements.

Case 1



Case 2



Case3

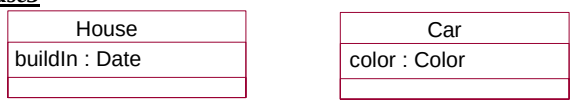


Figure 4 To generalize or not to generalize

Consider the purpose of the need for polymorphism. If the semantic concepts of an "account" or of "the balance of account" are real-world facts to represent, combined with the observation that savings accounts and currents accounts are accounts, both classes and the characteristic `balance` can be generalized into a class `Account`. At that time, the question arises if other model elements of the classes `SavingsAccount` and `CurrentAccount` should be generalized as well.

2.2 Anti-Patterns

From our perspective, common model elements of classes are no sufficient condition to generalize classes and their model elements. Consider the example outlined in Figure 5. In this example, the class `BankPerson` is added to the conceptual model in order to represent the common features `name` and `changeName()` only once. Instead of the class `BankPerson`, the analyst could also have introduced the class `NamedObject`, representing the set of all objects with a name. The introduction of this class would have violated the principle of abstraction. Not all objects with a name are relevant.

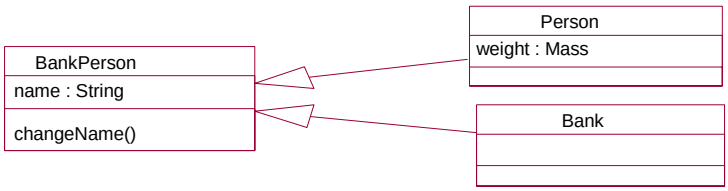


Figure 5 Common model elements are no sufficient condition to generalize

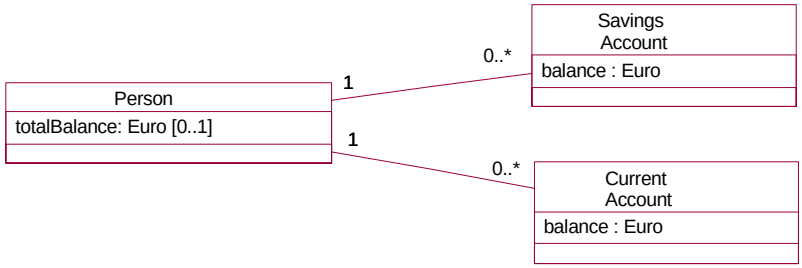
We agree with Borgida [Borg88, p.352], stating that the introduction of intermediate classes, whose only role is to act as anchors for inheritance, have uninteresting extents. Such intermediate classes make it harder for the users to find the meaningful classes. A primary goal of analysis is to represent relevant semantic concepts. In the example, a “bankPerson” is not a relevant semantic concept. We see inheritance as a consequence of generalization, not as a purpose.

2.3 Patterns

2.3.1 The Need for Polymorphism is the driving Force to generalize Model Elements and Classes

As stated in [Beka02, p.34], the need for polymorphism is our driving force to use the construct of a generalization relationship. The need for polymorphism manifests itself in the need to reason about a class as a concept representing a (super)set of objects, or in the need to reason about a model element at the level of a superclass. This means that there must be a semantic foundation to generalize classes or model elements of classes.

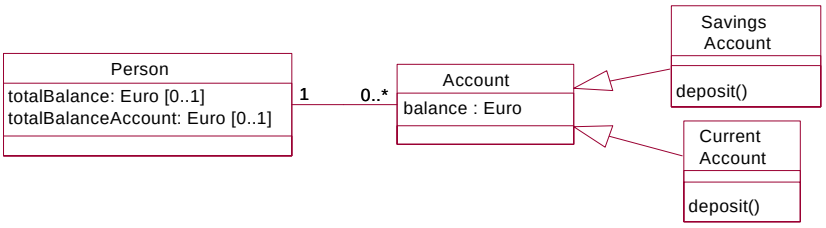
Consider the example outlined in Figure 6. Assume that there is a semantic concept “totalBalance”, which equals the sum of the balances of all the savings accounts and all the current accounts owned by a person. In this example, there is *no* semantic basis to generalize. The semantic concept of an account is *not* present. In order to model this dependency, there is no need to generalize. We claim that it is better to model the semantics of the stated dependency, and no more. This would not be the case if the dependency was modelled with the help of a superclass X and the subclasses `SavingsAccount` and `CurrentAccount`. If later on a new subclass of class `X` is added to the model, the semantics of the dependency changes. The semantic concept “totalBalance” would then equal the sum of the balances of the savings accounts, the current account and of the objects of the new subclass.



context Person **constraint:** totalBalance() =
savingsAccount().balance(),currentAccount().balance()→union()→sum()

Figure 6 No polymorphic need

Consider another example outlined in Figure 7. Assume the presence of a semantical concept "totalAccountBalance", which equals the sum of the balances of all accounts owned by a person. Presenting this case, the semantic concept of "the balance of an account" is the basis for generalization. Notice that the semantic concept of "totalBalance" of the previous example can be still present. If the subdivision is incomplete, which means that the union of sets of savings accounts and current accounts is a proper subset of the sets of account, then the concept of "totalBalance" differs from the concept of "totalAccountBalance". If the subdivision is complete, some redundancy is modelled. In such a case, the queries `totalBalance()` and `totalAccountBalance()` returns the same result for the same person. Recall that redundancy is allowed during analysis. This redundancy can disappear in the future, for instance if a new type of account is defined as another subclass of the class Account.



context Person **constraint:** totalBalance() = savingsAccount().balance(),currentAccount().balance()→union()→sum()

context Person **constraint:** totalAccountBalance() = account().balance()→sum()

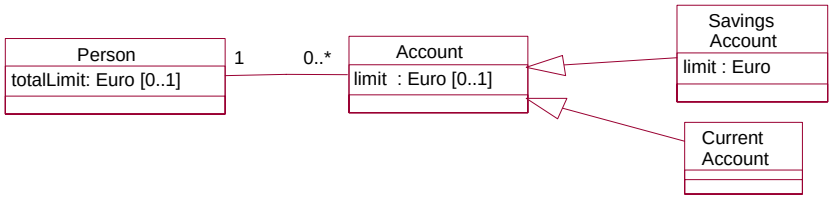
Figure 7 Polymorphic need

Common model elements of subclasses are not automatically generalized. This would violate the principle of abstraction. Reconsider the example of Figure 7. The event `deposit()` is not generalized because there is no polymorphic need. It this event would be generalized, the analyst is forced to investigate if the specification of the event at the level of the subclasses is also valid at the level of the superclass.

2.3.2 Common Model Elements are not a necessary Condition to generalize Classes

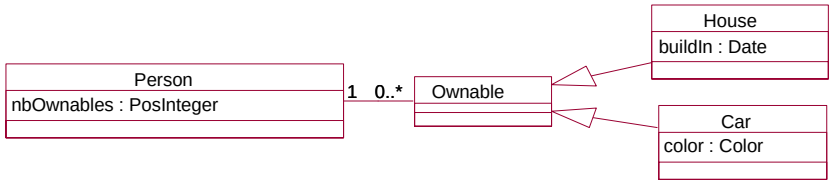
As already stated [2.2], common model elements of classes are not a sufficient condition to generalize. Moreover, they are not a necessary condition to generalize. Consider the examples outlined in Figure 8. In the first case, the characteristic `limit`, which was specified at the level of the subclass `SavingsAccount`, is generalized because the semantic concept `totalLimit` must be represented. Notice that the characteristic `limit` was not a common characteristic of both subclasses. In the second case, the class `Car` and `House`, which have no common model elements, are generalized because the semantic concept of "nbOwnables" must be represented. This semantic concept is defined as the number of objects a person owns.

Case One



context Person **constraint:** totalLimit() = account().limit()→sum()

Case Two



context Person **constraint:** nbOwnables() = ownable()→size()

Figure 8 Common model elements are no necessary condition to generalize

2.3.3 The Inheritance Relationship

On the one hand, we stated that sharing common model elements is not a sufficient condition to generalize classes. On the other hand, we recognize the technical need of an analyst to reuse specifications. Therefore we introduced the constructs of an ***inheritance relationship*** and the construct of a ***technical class***.

A technical class is only used to attach common model elements to it and does not represent a set of real-world objects. Such a class is hidden in the conceptual model and a name for it is not mandatory. Consider the example in Figure 9. A technical class, which is normally invisible in a conceptual model, is used to introduce the features `name` and `changeName()`.

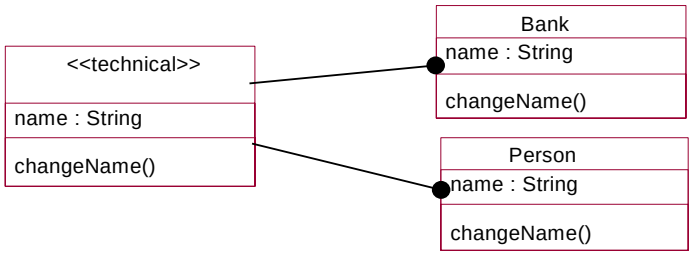


Figure 9 Technical class and a pure inheritance relationship

Between a normal class and a technical class an inheritance relationship can be defined. An inheritance relationship is denoted by a line with a filled circle. An inheritance relationship does not express set inclusion relationship and is only used for pure inheritance purposes. The subclass inherits all model elements of the technical class. In the example of Figure 9, two pure inheritance relationships are introduced between the technical class on the one hand and the class `Bank` and the class `Person` on the other hand. The classes `Bank` and `Person` inherit all model elements introduced at the level of the technical class. These inherited model elements are shown in both classes. Recall that there is no semantic need for an inheritance relationship. As one of my eloquent colleagues said, the need for a pure inheritance mechanism is nothing more than the need for an intelligent copy and paste.

The introduction of both constructs also decreases the need to introduce the construct of exceptions as proposed by Borgida [Borg88]. Borgida's approach allows explicitly acknowledged exceptions in class definitions, enabling a subclass to state that it will not inherit some model elements from its superclass. A well-known example in artificial intelligence in which this approach can be used, is that of all birds can fly with the exception of penguins, which do not fly.

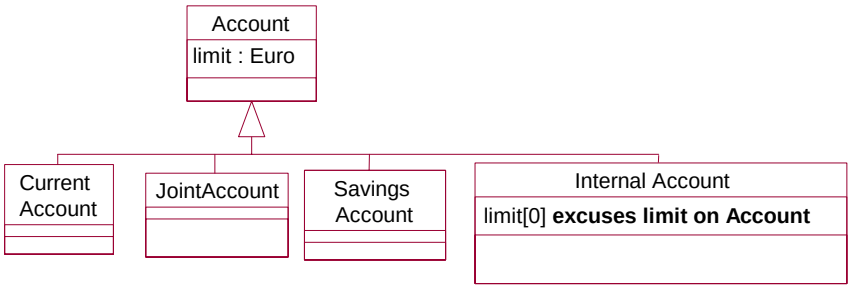


Figure 10 Approach with exceptions

Consider another example outlined in Figure 10. Assume that all accounts have a limit, except for the internal accounts of a bank. In the class `Account`, a characteristic `limit` is defined. In the approach with exceptions, this characteristic reflects that in principle each account has a limit. In the class

`Internal Account`, it is specified that, contrary to accounts in general, internal accounts have no limit. Borgida explicitly specifies exceptions. The other subclasses inherit the characteristic `limit`. Borgida claims that this approach maintains the benefits of inheritance and polymorphism.

The construct of exceptions is promoted by arguing that without this construct, the analyst encounters problems to represent real-world facts. Each of these problems is described below:

- The analyst looses the benefits of inheritance in case the characteristic `limit` is not defined in the class `Account` but in all the subclasses, except in the subclass `InternalAccount`.
- The analyst is forced to introduce intermediate classes such as `AllAccountsExceptInternal`, which represent an uninteresting set of objects, in case the analyst wants to preserve the benefits of inheritance.
- The analyst looses the benefits of polymorphism in case the approach allows contradictions in the specifications of subclasses and superclasses.

With the introduction of technical classes, the analyst preserves the benefits of inheritance and can avoid the presence of intermediate classes in the conceptual model without a need for the construct of exceptions. Furthermore, contradictions are still not allowed in this approach and therefore the benefit of polymorphism is maintained. Consider the approach with technical classes in Figure 11. The characteristic `limit` is defined in the technical class and pure inheritance relationships are defined between all subclasses except for the subclass `InternalAccount`.

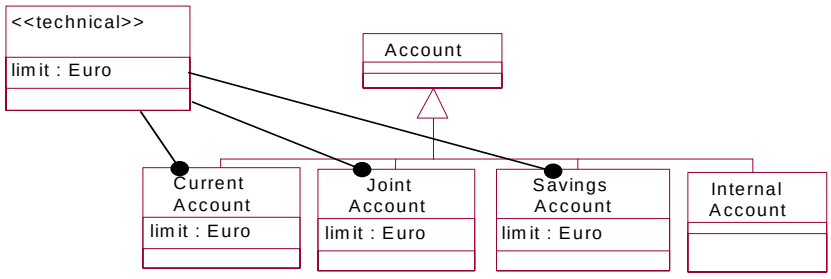


Figure 11 Approach with technical classes

3 Modelling Particulars

3.1 Problem Description

A ***particular*** is defined as something that is separate and distinct from others of the same group or category. In object-oriented analysis, we define a particular as a property or restriction that is not valid for all objects of a certain set of objects. Consider the following example. A bank, with the name “KBC”, has a business rule, stating that the balance of an account must at all times exceed the limit of that account. Notice, that this business rules applies to accounts belonging to the KBC-bank and that it is not certain that this business rule also applies to other accounts. In fact, it is not even certain that all accounts have a limit for their balance. Furthermore, it is possible to transfer money from one bank account to another bank account. A transfer is not limited to accounts of the KBC-bank. From the viewpoint of the set of accounts and banks, the name of the bank, the stated business rule and the property limit are particulars.

Typically, the taskmaster of a software development project is an organization. Some features are only valid for objects involved in this organization. Some features have a broader range. Such features are valid for objects involved in the domain in which the organization is active. How can these real-world facts be modelled in an adequate way?

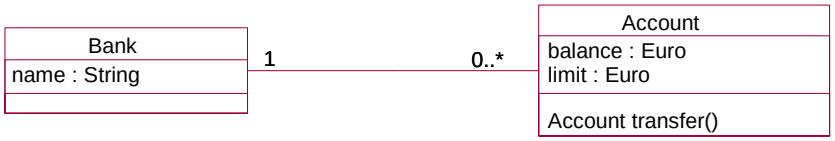
An often-occurring problem is that of determining the class of a feature, which represents a particular property or restriction. Such a feature can be attached to a more generalized class or to a more specialized class. Ultimately, a particular can be only valid for one object. The name of the KBC-bank and the fact that KBC-bank is the only bank where one can open KBC-accounts are examples of particulars, which are only valid for one object.

3.2 Anti-Patterns

3.2.1 Adopting a Universal Approach

In a universal approach, the analyst attaches features at the level of more generalized classes. Consider the alternatives in Figure 12 and in Figure 13. In both alternatives, the features are attached to the more generalized classes `Account` and `Bank`. The class `Account` models the set of all bank accounts; the class `Bank` represents the set of all banks.

In both alternatives, the name of the KBC-bank is used to specify some class constraints. The stated constraint implies that the name of the "KBC-bank" is unique and that bank will not change its name or stop its activities. Obviously, these modelled facts do not correspond with reality. The name of the bank is misused to identify that bank. One could argue to change the first class constraint into `Bank.Allinstances()→select(name()="KBC")→size()<= 1`. This constraint would allow a bank with the name "KBC" to stop its activities, but the existence of a bank with the name "KBC" in the real world would not be enforced by this constraint.



context Bank **constraint:** Bank.Allinstances()→select(name() = "KBC")→size() = 1

context Account **constraint:** balance() >= limit()

context Account,Account :: transfer(amount: Euro)
eventc: self1 <> self2
effect: self1.balance() = self1.balance() + amount
self2.balance() = self2.balance() - amount

Figure 12 Attaching features to more generalized classes (1)

In the alternative outlined in Figure 12, the principle of abstraction was violated. The analyst had to investigate whether the business rule concerning

the balance applies to all other bank accounts, and whether each bank has indeed a name. This could be the case, and then the first-observed particulars aren't particulars anymore. In the other case, there is no answer on how to model particulars. We do not want to force the analyst to investigate if certain features also apply on a bigger set of objects, if for the sake of the conceptual model it is sufficient to describe the facts that apply to a particular subset.

context Bank **constraint:** Bank.Allinstances()→select(name() = "KBC")→size() = 1

context Account **constraint:** bank().name() = "KBC" implies limit() →size() = 1
context Account **constraint:** bank().name() = "KBC" implies balance() >= limit()

context Account,Account :: transfer(amount: Euro)
eventc: self1 <> self2
effect: self1.balance() = self1.balance() + amount
self2.balance() = self2.balance() - amount

Figure 13 Attaching features to more generalized classes (2)

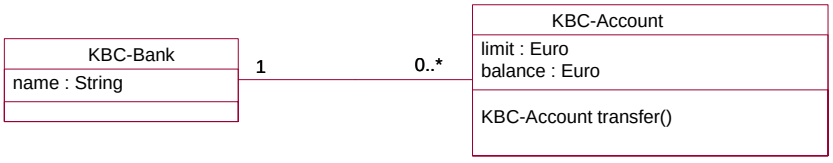
In the alternative outlined in Figure 13, the features are also attached to more generalized classes but the principle of abstraction is more or less satisfied. The analyst didn't investigate whether each account has a limit, whether each bank has a name, nor whether the business rule concerning the balance applies to other accounts than KBC-accounts. Notice however that the principle of abstraction is not fully satisfied. The analyst did investigate that accounts can have at most one limit, that banks can have at most one name, and that the possible limit of an account is expressed in Euro. In order to satisfy the principle of abstraction, the multiplicity of the characteristics `name` and `limit` and the data type of the characteristic `limit` could be abstracted. In trying to satisfy the principle of abstraction, this approach leads to more and more complex constraints. Compared with the previous alternative, the second class constraint must be added and the third class

constraint is more complex. As in the previous conceptual model, there is no polymorph need to generalize the characteristics `name` and `limit`.

3.2.2 Adopting a Particular Scope

In a particular approach, the analyst attaches features at the level of more specialized classes. Consider the example outlined in Figure 14. In the example, the features are attached to more specialized classes such as `KBC-Account` and `KBC-Bank`. The class `KBC-Account` represents the set of all accounts of the KBC-bank; the class `KBC` represents the set of all KBC-banks. The last set is a singleton; it contains only a single object.

The adoption of a particular scope looks attractive at first sight. The model is strictly limited to the objects of the particular scope. The analyst can focus on modelling properties and constraints of these objects without concerns if these features are also present for other similar objects in the universe. Consequently, features of a particular organisation can be modelled without violating the principle of abstraction or without leading to more and more complex class constraints. However, the problem is that the scope can be too narrow: not all relevant real-world facts are then represented. In the example, one can only transfer money from one KBC-account to another KBC-account and not to other accounts. The narrow scope violates the principle of completeness because not all relevant real-world facts have been represented.



context KBC-Account **constraint:** balance() >= limit()

context KBC **constraint:** KBC.Allinstances()→size() = 1
context KBC **constraint:** name() = “KBC”

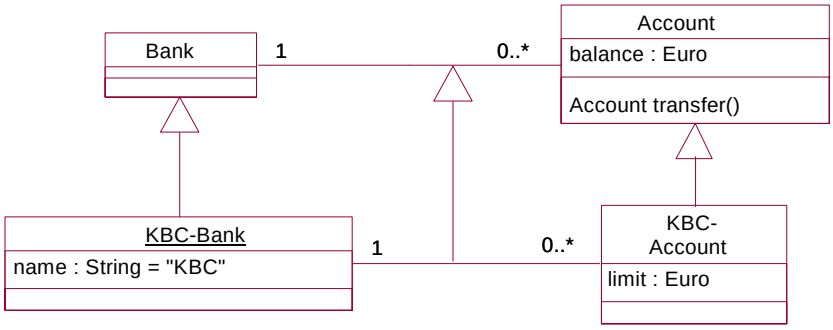
Figure 14 Attaching features to more specialized classes

Notice that the problems concerning the name of bank, mentioned in subsection 3.2.1 are still valid. The KBC-bank cannot change its name or stop its activities. One could argue to specify a characteristic `name : String = "KBC"` instead of a class constraint `name() = "KBC"` to model the name of the bank. However, the construct of specifying the initial value of a characteristic does not bring a solution. The initial name of the bank is specified, but no statement is made about the current name of the bank.

3.3 Patterns

3.3.1 Adopting a mixed Scope

In adopting a mixed scope, the analyst attaches some features to more specialized classes and others to more generalized classes without violating the principles of abstraction and of completeness. This approach will also not result in more and/or more complex class constraints. Consider the example outlined in Figure 15. The event `transfer()` is attached to the class `Account`. If the event would be attached to the class `KBC-Account` the principle of completeness would be violated. The class constraint `balance() >= limit()` on the other hand, is attached to the class `KBC-Account`. If this constraint would be attached to the class `Account`, the principle of abstraction would be violated or a more complex specification of the constraint would be needed such that it only applies to KBC-accounts.



context KBC-Account **constraint:** balance() >= limit()

context Account,Account :: transfer(amount: Euro)
post: self1.balance() = self1.balance() + amount
self2.balance() = self2.balance() - amount

Figure 15 Adopting a mixed scope and introducing objects

The "KBC-bank" is represented as an object and not as a class. The name of an object is underlined. The construct of objects is further worked out in the next subsections.

3.3.2 Objects in a Class-Diagram

In some cases, there can be a need to model particulars of one single object instead of a set of objects. For instance, the KBC-bank is selling KBC-accounts and it is the only bank that sells such accounts. Obviously, this feature cannot be generalized for other banks. In order to model such a particular, objects are introduced in the class diagram. Underlining their name distinguishes objects from classes in conceptual models. In the example, the object KBC-Bank is introduced and represents the existence of the KBC-bank in the real world. This object is necessary in order to model the presence of the object in the real world and to model the particulars of this object. An object is the smallest possible range an analyst can focus on.

One could argue that the modeller could have used singleton classes instead of objects. The problem with this kind of classes is that classes represents sets of objects, in which objects can enter or can leave. This is not the desired semantics for a particular object: no other objects can enter another object. That is the main reason why it is preferred better to model objects as such, and not as singleton classes. Moreover, with objects one can avoid explicit class constraints, which restrict the number of elements in a set. Obviously, no constructors can be attached to an object.

In the UML [OMG01, p.3-35], object diagrams are defined as class diagrams with objects and no classes. Moreover, a class diagrams can contain objects and classes. The use of objects and object diagrams is fairly limited. They

are mainly used as illustrations of possible populations of classes and class diagrams. We do no object against using objects for purposes of illustration. We do believe that the construct of an object is very useful to model particulars and to reflect the existence of core objects in the external world. In the UML, an object is shown as a rectangle with two compartments. The first compartment shows the name of the object and its class. The second compartment shows the attributes of the object and their values. In the UML, these two compartments are sufficient, because objects are only used for illustration purposes, and not as a key construct to model particulars. As illustrated in the example, we suggest representing objects in the form of rectangles with three compartments. The third compartment is used to represent particular dynamic properties.

Notice that the specialized association [Rumb99, p.163] between the class KBC-Account and the object KBC is necessary; otherwise, KBC-accounts could be linked with other banks. This specialized association refines the association between the class Account and the class Bank. Accounts are linked with a bank and KBC-accounts are linked with the KBC-bank. Both constructs of the specialized association are indispensable: the association and the generalization relationship. If the generalization would be omitted, a KBC-account would have a link with the KBC-bank and would have an additional link with a bank. As we stated in [Beka02, p.37], specialization of associations is practised rather rarely and considered to be exotic, while it has a high potential and can be used quite often.

3.3.3 Object Generalization

If an object is an instance of a class in the conceptual model, then a generalization relationship between the object and its class is drawn. In the example, a generalization is modelled between the object KBC-bank and the class Bank. Obviously, it is impossible to further specialize an object. The introduction of objects in a class-diagram gives the analyst the opportunity to fit the particular in the universal.

In the UML, the name of the class of an object is indicated in the object itself because objects are used primarily in object diagrams. This is another

indication that the UML has introduced objects for purposes of illustration, and not as an integral part of class diagrams. In our approach, it is also possible that objects are not generalized. In the UML, each object is an instance of a known class.

3.3.4 Object Characteristic

An object characteristic allows representing the link between an object and a data value. An object characteristic is a characteristic attached to an object. In the example of Figure 15, the characteristic `name: String = "KBC"` indicates the name of the KBC-bank. Notice that this specification does not specify an initialisation value or a constraint. The bank can later on change its name. In the UML, a similar construct is present [OMG01, p.3-65]. However in the UML, the data type is not specified because the class of the object is always present. The data type cannot be further refined, because objects only serve for purposes of illustration. As a consequence, no new or specialized information can be introduced at the object level. In our approach, the data type of an object characteristic can be specified, because its class will not always be included in the conceptual model. In addition, the data type of an object characteristic can be further refined at the level of the object itself.

Chapter 7

Patterns for Modelling Requirements

Analysis does not only concern the representation of the relevant real-world facts, but also deals with a specification of the requirements [Hoyd93, p.242]. The requirements specification is the description of the ultimate expectations, objectives and needs of the users towards the system. Imprecise or incomplete requirements specification is often cited as a major reason for system project failure [Gree94, p.135]. Misunderstandings during the requirements specification are reproduced during the whole development cycle. Consequently, rectifications of such misunderstandings can be costly. The crucial place of requirements specification in the software development cycle is nowadays generally recognized.

Requirements can be split up into functional requirements and non-functional requirements. Functional requirements are further discussed in the first and only section of this chapter. Non-functional requirements deal among other things with aspects such as response time, ease of use, fault tolerance, platforms, security and budgeting. As stated in [Roll92, p.4], the non-functional requirements are often expressed in an unstructured way. Research efforts should try to formalize non-functional requirements.

1 Modelling Functional Requirements

1.1 Problem Description

Functional requirements specification is a description of the desired functionality of the system to be built. Consider the following examples of functional requirements of a system to be built for a bank: (1) the system should, given the account number and some amount of money, deposit the given amount to the account with the given number; (2) the system should, given a bankcard, present the balance of the account to which the given

bankcard applies; (3) the system should on demand present the total balance of the “KBC” bank.

How should these functional requirements be specified? We fully agree with Jackson [Jack84, p.4] where he eloquently promotes the fundamental principle that the developer must begin by modelling the reality and only then go on to specify the functionality of the system. Like Jackson, we are convinced that a great part of the complexity of a system is caused by the complexity of the real world. By separating the process of specifying the functional requirements from specifying the domain knowledge, the analyst is able to focus in a first phase on the real-world facts. As stated in [Snoe99, p.14], software engineers tend to mix these specifications, together with technical requirements.

Furthermore, the specification of the functionality should be built on top of the specifications of the observed real-world facts. This ensures that the functional requirements are unambiguously specified in terms of the observed real-world facts [Jack84, p.8]. In this way, the conceptual model serves as a context for the specification of the system functions.

1.2 Anti-Patterns

In the UML, the functionality of the system to be built is mostly expressed in terms of use cases [Kost01, p.3]. A use case is defined as a coherent unit of externally visible functionality provided by a system unit and expressed by a sequence of messages exchanged by the system unit and one or more actors of the system unit [Rumb99, p.64]. Although the UML indicates that a use case can be described in several different ways [OMG01, p.3-96], it is a widespread practice that uses cases are primarily textual descriptions. Aside the textual description, a use case diagram can be built, but such a diagram only represents the names of the use cases and the actors together with their relationships [OMG01, p.3-94].

Assume a system to be built for a shop. The system should help to checkout customers buying items, and should perform sales analysis and inventory control. In order to evaluate use cases as a way to capture functional requirements, an already existing example of a use case will be discussed.

This use case, with the name “buy items”, describes the interaction of the actor with the system in case the customer checks out. The system should present the sale total, the balance due to the customer and it should produce a receipt for the customer. Consider the use case diagram in Figure 1 and the textual description of a use case in Table 1. This example is based on an essential use case described in [Larm98, p.67], but it is slightly adapted.

An essential use case, which is the opposite of a real use case, is defined as a use case that remains relatively free of technology and implementation details. Design decisions are deferred and abstracted, especially, those related to the user interface [Larm98, p.58]. In the example, abstraction is made of the precise identifier for an item and of the kind of user interface.

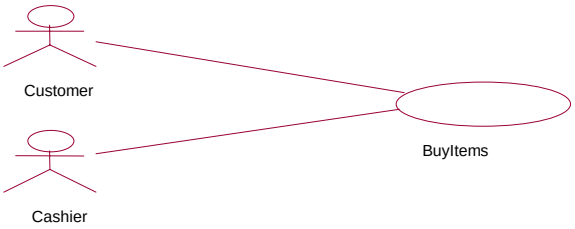


Figure 1 A use case diagram

Actor Action	System Response
1. This use case begins when a customer arrives at a checkout of the store with items to purchase.	
2. The cashier records the identifier for each item. If there is more than one piece of the same item, the cashier can enter the quantity as well.	
3. On completion of item entry, the cashier indicates that the item entry is complete.	4. Calculate and present the sale total.

5. The cashier informs the customer about the sale total.	
6. The customer chooses for a cash payment.	
7. The cashier records the cash received amount.	8. Show the balance due back to the customer and generate a receipt.
	9. Log the completed sale.
	10. Update inventory levels
11. The cashier deposits the cash received and extracts the balance owing. The cashier hands over the balance owing together with the printed receipt to the customer.	
12. The customer leaves with the items purchased.	
Alternative Courses 2. Invalid identifier entered. Indicate error. 7. Customer could not pay. Cancel sales transaction.	

Table 1 Textual desciption of a use case

We recognize that with essential use cases an effort is made to avoid design decisions. However, we believe that the above approach to capture and represent functional requirements has several disadvantages:

- A first disadvantage is that building uses cases is usually considered to be a first step in the software engineering process, see for instance, [Fowl97, p.51] and [Kost01, p.3]. This violates Jackson's fundamental principle that the developer must begin by modelling the reality and only then go on to consider in detail the functionality of the system.

- One could argue that the technique of use cases can be used after building the class diagram, which models reality. This is correct, but a second disadvantage is that there is no close connection between both modelling techniques. Ideally, the specification of the functional requirements should be built on top of the specifications of the domain model. As stated in [Kost01, p.3], requirements specification mainly deals with use cases and class models, but unfortunately these models are based on

different modelling techniques and aim at different levels of abstractions, such that serious consistency and completeness problems are induced. In the same paper, a traceable transition of uses cases to a class model is presented. In [Send00], techniques to map use cases into system operation specifications are highlighted. These papers all try to link use cases with other models, but both papers still suggest to start with the use cases.

- A third disadvantage is that use cases are (normally) informal. This violates the principle of preciseness. One could argue that this turns uses cases into a popular and adequate technique for communication with non-technical stakeholders. This could be correct, but notice that we make no statements about how an analyst should communicate with his stakeholders. We consider the proposed conceptual models not *necessarily* as a communication tool. If a particular stakeholder doesn't understand models, the analyst shouldn't use these models as a way to communicate with this particular stakeholder. This doesn't mean that the analyst must not longer specify real-world knowledge in a conceptual model.

- In use cases, no abstraction is made of the actors. As already stated in chapter 4, we want to make abstraction of the users during the specification of functional requirements. Reconsider the second line in Table 1: according to the use case, it is the cashier who records the identifier of each item. During the specification of the functional requirements it is irrelevant who records the identifier of the item; it could also be the customer who performs "self-scanning".

- Use cases are too much focused on the dialogue between the actor and the system. This could be an interesting focus when user interfaces or system operations are designed, but not when functional requirements should be specified. This focus necessarily introduces design decisions even if an effort is made to avoid them. Reconsider the description of the use case in Table 1:
    - Line 2: if there is more than one of the same item, the cashier can enter the quantity as well. This is a user interface design decision.

  - Line 9 and line 10: a sequence between logging the sale and updating the inventory is already determined and it is already decided to store information.
  - Alternative course line 2. This is a user interface design decision. Another user interface could be considered in which it would be simply impossible to enter an invalid identifier.
  - Line 3: this is once again a user interface decision. Another user interface could be considered where the item entry is automatically completed when no items are entered for a certain period of time.

- Contrary to popular belief, see for instance [Mala01, p.1], use cases are not primarily goal-oriented. The focus of use cases is on system interaction and can hide the user goals. This is also noticed in [Fowl97, p.44], where an example is given of uses cases for a word processor as "define a style" and "change a style". This hides the user goal of "ensure consistent formatting for a document". We introduce another example. Consider a system to be built that should return the height of a person given his social security number. If a designer decides to build a database containing social security numbers and heights of persons, a possible use case is "the registration of personal data". This use case could also be considered as a functional requirement: the system should perform the registration of personal data. If one starts with the description of use cases, the goal of the system is hided: the goal of the system is not to register personal date. If another designer decides to connect the system to be built with another system, which contains the personal details, no registration is needed. As already observed, uses cases already imply design decisions. This example is further discussed in the next subsection.

- Use cases take real-world facts into account. Often, several use cases have to take the same real-world facts into account. The same real-world facts are then typically described in several use cases. This does not promote adaptability. Consider the well-known business rule "balance exceeds limit". The use case "withdraw", the use case "transfer" and the use case "modify limit" will describe a in certain way this restriction and the possible consequences (cancel withdraw, cancel transfer and cancel modify limit). In the UML [Booc99, p.227], an include relationship between use cases can be defined. Such a relationship is used to avoid

describing the same flow of events, but the extreme use of this construct would lead to very fine-grained functional decomposition. For example, every business rule should then be modelled as an included use case "checking business rule".

- A use case is not an object-oriented analysis artefact [Larm98, p.10]. It is more oriented towards functional decomposition than to object-orientation. It can be equally (in)effective and applied in a non object-oriented development. Consider line two as an example. Abstraction is made of the precise identifier for an item. One could argue that the decision about the way an item is identified is a design decision and should be postponed. This is correct, but a typical aspect of object-orientation is that each object is unique and has an implicit identity. It is preferred to make use of this implicit identity instead of using an abstract identifier.

- No strict guidelines exist about the way to build use cases [Send00, p.2]. As an example, one could complement the above use case with a description that it is possible to modify the entered identifiers and the entered quantities until the completion of the entries. Should this possible modification be described in the use case or not?

As a conclusion, we do not consider use cases as an adequate technique to capture and represent functional requirements due to the disadvantages described above. Notice that we make no statements about the adequateness of use cases as a communication tool with a stakeholder, nor as a technique for establishing in detail the communication between the end-users and the system.

1.3 Patterns

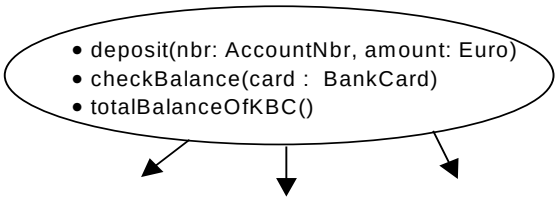
1.3.1 The Functionality Layer

We propose to specify the functional requirements with the help of a functionality layer. The conceptual model is then divided into two layers: a ***domain layer***, in which the real-world facts are represented, and a ***functionality layer***, in which the functional requirements are specified. The

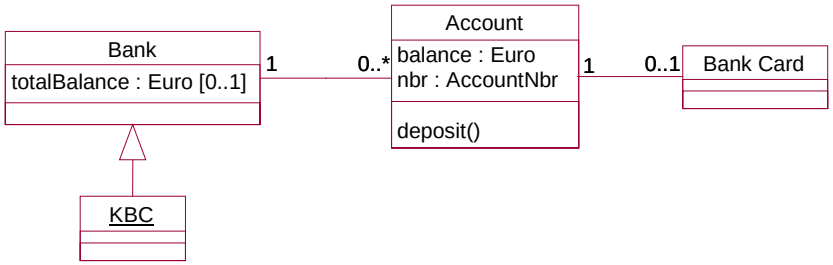
specifications of the functional requirements are built on top of the specifications of the domain layer.

As an example, assume the following real-world observations. Each account has a balance and an account number, and each account can have at most one bankcard. A deposit on an account increases the balance of that account with a given amount. The total balance of a bank equals the sum of all balances of its accounts. The "KBC" is a bank. These real-world observations are represented in the domain layer of Figure 2. Functional requirements can be built on top of these real-world observations. Assume that a system is needed that increases the balance of an account with a given amount. The account is identified through its unique number. Furthermore, the system should present the balance of an account given its associated bankcard, and it should present the total balance of the "KBC" bank on demand. These requirements are modelled in the functionality layer.

Functionality Layer



Domain Layer



-- specifications functionality layer
context functional requirement :: deposit(nbr: AccountNumber, amount : Euro)
post: Account.AllInstances()→select(nbr()=nbr).deposit(amount)

context functional requirement :: checkBalance(card : Bank Card) : Euro

post: result = card.account().balance()

context functional requirement :: totalBalanceOfKBC() : Euro
post: result = KBC.totalBalance()

-- specifications domain layer
context Account :: deposit(amount : Euro)
effect: balance() = balance()@pre + amount

context Bank **constraint:** totalBalance() = account().balance()→sum()

Figure 2 Modelling the functional requirements in a separate layer

We believe that there is a fundamental distinction between real-world facts and the functional requirements of a system to be built. The first is about observations of the world around us, and the second is about the functionality of a system. Specifying that each account has a balance and can have a bankcard is something else than specifying that a system should present the balance of an account on the basis of its bankcard. It is not because a characteristic `balance` is attached to the class `Account` that a requirement is represented that returns the balance of a given account. The layered conceptual model makes this fundamental distinction between requirements and real-world facts explicit.

In EROOS [Steeg00] and other analysis methods [Roll92, p.4], the functional requirements are integrated in the conceptual model: no explicit distinction between real-world facts and functional requirements is made. In EROOS, more complex queries and events are said to specify the functional requirements. From our point of view, more complex characteristics and events, as for instance a transfer of money, represent real-world facts. The layered conceptual model separates the specification of the functional requirements and the specification of the real-world facts. This enables the analyst to focus in a first phase on real-world aspects, which is inline with Jackson's fundamental principle. If it is fundamental to first model the real-world facts and after that the requirements, then it is essential to make a distinction between facts and requirements.

The analysis method MERODE also introduces a functionality layer [Snoe99, p.205]. However, this layer is not only used to model requirements

but it is also used to model more complex real-world facts. Only atomic non-decomposable events are allowed in the domain layer [Snoe99, p.26]. The functionality layer is also used to represent more complex events. In the domain layer of our conceptual model, it is not because an event is decomposable that it is not a real-world fact. We will only use the functionality layer for modelling functional requirements.

The pattern described in this section is about layering conceptual models and about the distinction between functional requirements and real-world facts. The pattern doesn't describe in detail how to specify and structure the functional requirements. All other patterns in this text only apply to the domain layer. For instance, messages modelling the functional requirements can have objects as explicit arguments and these messages are not attached to a class. We underline that not the notation, in our case an oval, is important but the method, the segregation of responsibilities in hierarchical layers. Further research is needed on how to specify and structure functional requirements.

1.3.2 Other Advantages of the Functionality Layer

The functionality layer can be used as a way to control whether all relevant domain knowledge is present. If it is impossible to specify a functional requirement, then the domain layer should be extended. Reversely, the functionality layer can also be used to control if all domain knowledge is relevant. If some domain knowledge is nowhere used in specifications of functional requirements, then it is irrelevant.

The domain layer is independent of the functionality layer. It cannot make use of specifications worked out in the functionality layer. It should be possible to represent real-world facts without having a system and requirements in mind, for instance, for pure business modelling purposes where only the business as such should be represented and where no system development is envisaged. A positive consequence is that the adaptability of the conceptual model is stimulated. The domain layer is considered to be more stable than the functionality layer [Snoe99, p.14]. The functional requirements tend to change more often than domain knowledge. An

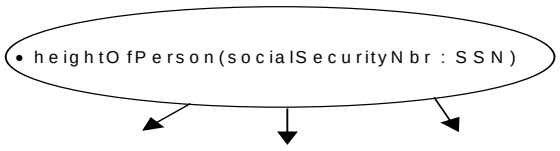
independent domain layer ensures that possible modifications in the functionality layer have no effect on the domain layer.

One could argue that instead of modelling a functionality layer, the analyst could have introduced a class, which represents the users of the system. The functional requirements could then be attached to this class. The disadvantages are that no abstraction is made of the possible users of the system and that no distinction between requirements and real-world facts is made. Another disadvantage is that all of the messages of the functional layer are associated with this class. This class would be a receptacle of different kinds of messages.

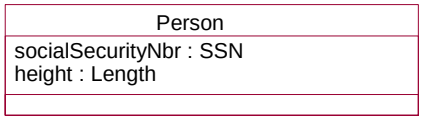
1.3.3 Real-world Objects

A misunderstanding could be that from the moment the functional requirements are modelled, the analyst is working with software objects instead of real-world objects. This is not the case. The objects represented in the conceptual model are still real-world objects. Consider the conceptual model outlined in Figure 3. Assume a system that should return the height of a person, given his social security number. In the functionality layer, the analyst cannot use a constructor of the class `Person` to model the registration of personal data in the system, because the constructor for persons represents their birth and not the registration of personal data in an information system. In other words, the requirement of an information system to register personal data cannot be modelled. From our point of view, this is not a disadvantage but an advantage. The registration of a person is only the consequence of a design decision to build an information system. Other systems can be imagined where it is not necessary to register personal data. If this would be modelled, a premature design decision would have been taken.

Functionality Layer



Domain Layer



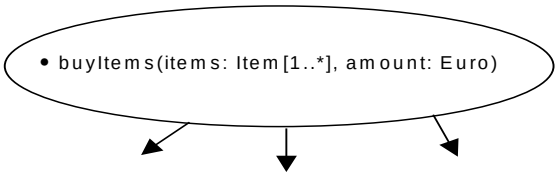
-- specifications functionality layer
context functional requirement :: heightOfPerson(socialSecurityNbr : SSN) : Meter
post : Person.AllInstances()→select(socialSecurityNbr()=socialSecurityNbr).height()

Figure 3 In the conceptual model real-world objects are represented

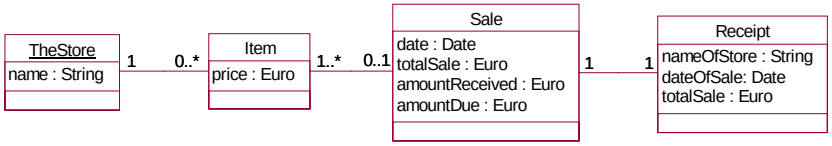
1.3.4 A last Example

Consider a last example in Figure 4, where a functional requirement "buy items" is specified. The example is based on the use case described in section 1.2. The system should help to checkout a customer. The system should present the sales total and the amount due to the customer, given the set of items the customer wants to buy and the received amount. The system should also generate a receipt. In contrast with use cases, the requirement in the functionality layer is formally specified and the focus is not on system interaction. Abstraction is made of possible actors.

Functionality Layer



Domain Layer



```
-- specifications functionality layer
context   functional requirement :: buyItems(items [1..*]: Item, amount : Euro) :
          receipt : Receipt, totalSale : Euro, amountDue : Euro

post :    let newSale = (Sale.allInstances() – Sale@pre.allInstances()) -- new Sale
          newSale→size() =  1 and newSale.oclIsnew()
          let newReceipt=(Receipt.allInstances()–Receipt@pre.allInstances()) -- new Receipt
          newReceipt→size() =  1 and newReceipt.oclIsnew()

          newSale.item() = items and newSale.amountReceived() = amount
          newSale.date() = now()  and newSale.receipt() = newReceipt

          receipt = newReceipt – returned results
          totalSale = newSale.totalSale()
          amountDue = newSale.amountDue()

-- specifications domain layer
context   Sale constraint: amountReceived >= totalSale()

context   Sale constraint: amountDue() = amountReceived() - totalSale()
context   Sale constraint: totalSale() = item().price()→sum()

context   Receipt constraint: nameOfStore() = TheStore.name()
context   Receipt constraint: dateOfSale() = sale().date()
context   Receipt constraint: totalSale() = sale().totalSale()
```

Figure 4 A last example

In the functionality layer, data values as well as real-world objects are used as input and output for the system. In the example, a set of items is used as input for the system, while a real-world object as a receipt is specified as output. Notice in particular that the different items are not identified through their name or some other identifying characteristic.

In [Larm98, p.95], it is argued not to model the class `Receipt` because it is not useful since all its information is derived from other sources, although it is admitted that a receipt is a relatively prominent concept in the problem domain. As already argued, it is not because a concept can be derived that it cannot be present in a conceptual model. More important is that in the proposed method no doubt can exist whether or not this class should be left out. If the analyst wants to specify a requirement stating that a receipt must be generated, this receipt must be included in the conceptual model. Notice

also that the conceptual model specifies in a detailed way which information should be presented on a receipt.

In the model, no decisions about the user interface are already taken. Moreover, no decisions are taken about which information of the sale is stored or about the update of the inventory. This thus not prevents the analyst to specify functional requirements on the existing domain layer about sales analysis and about the inventory.

The sales transaction will be cancelled if a client cannot pay due to the class constraint `amountReceived() >= totalSale()` in the domain layer. The sales transaction will also be cancelled if items that a customer wants to buy were already involved in another sale. The multiplicity constraints of the association between the class `Item` and the class `Sale` prevents that an item is sold more than once. Such real-world facts are only represented once in the domain layer.

We have to admit that we are not completely satisfied with the way the postcondition of the functional requirement is specified. The reason is that not one or two objects are involved in an event `buy()` but a set of objects, namely items. No pattern exists which describes how to model an event in which a set of objects is involved. Perhaps this event can be modelled by introducing an n-ary event `buy()` attached to the class `Item`, where n is an arbitrary number of items.

Chapter 8

Conclusions

In this thesis, we identified a number of important problems about object-oriented analysis such as a vague border between analysis and design, the need of adequate analysis constructs, adequate analysis guidelines and adequate analysis quality criteria and a loose coupling between the analysis results and the design activities. In section 1, the most important contributions of this thesis to solve some of the problems in object-oriented analysis are summarized. Section 2 describes the work that still should be done to solve other problems in object-oriented analysis.

1 Contributions

Firstly, the UML was evaluated as a language for conceptual modelling purposes. Secondly, our focus was on developing patterns to provide solutions for recurring conceptual modelling issues. Finally, we also defined principles that help to evaluate conceptual modelling constructs and guidelines.

1.1 Evaluation of the UML for Conceptual Modelling

We do *not* consider the UML in its present form, as an appropriate modelling language for precisely and completely representing observed facts in the real world. In this thesis, we rejected the UML constructs and semantics of derived attributes, attributes, queries attached to classes, invariants, preconditions, postconditions and use cases for analysis purposes. The reasons are threefold and explained in the next paragraphs.

The main reason is that the same modelling constructs and the same semantics for these constructs are offered by the UML to fulfil analysis

purposes as well as design purposes. If a construct and its semantics are well suited to describe a technical aspect about some software object, this construct and its semantics are by definition not suited to describe an aspect of a real-world object because this technical aspect is not relevant for the real-world object. We claim that the UML is far more suited for design purposes than for analysis purposes.

A second reason is that the offered modelling constructs not always support the principles of conceptual modelling as defined in chapter 1. For instance, if preconditions, implications and invariants are used, the principle of uniqueness is violated because a business rule is represented at different places.

A third reason is that deficiencies, such as construct redundancy, occur if the UML is used as a language for conceptual modelling. For instance, the constructs of a query and of a derived attribute are redundant at the level of analysis, which is not the case at the level of design. Worse, some constructs are not only redundant, but tend to confuse the mind of software engineers during conceptual modelling. Indeed, with constructs such as derived attributes, and even attributes as such, software engineers start thinking about design issues (representation), instead of keeping their focus on just describing real-world facts.

1.2 Patterns for Conceptual Modelling

In the next paragraphs, the most important patterns and their advantages are summarized:

- *Patterns for modelling static properties.* We proposed the introduction of the construct of a *characteristic* to represent a set of links between data values and objects. Derived attributes, attributes and queries were rejected as adequate constructs for conceptual modelling. In contrast with an approach that uses the rejected constructs, the approach with characteristics promotes the principle of extendibility, the principle of uniqueness and the principle of no-choice. This approach also avoids construct redundancy. Furthermore, possible confusion between the use of the rejected constructs during analysis and during design is avoided.

We reserve the rejected constructs exclusively for the design phase. *Encapsulation during analysis* and *modelling dependencies on the model layer* were introduced as guidelines to improve the adaptability of conceptual models.

- *Patterns for modelling dynamic properties.* In contrast with the OCL, we allowed the specification of *compound events*. A compound event allows expressing the semantics of an event in terms of other events. The effect of a compound event can always be reformulated in terms of queries. Another classification was made in terms of *mutators* and *migrants*. A mutator models a change of an object's link. A migrant models the migration of an object from a certain set into another set. We claimed that there is no logical necessity to forbid dynamic classification during conceptual modelling. *Frame axioms* are implicitly added to the effect clause. The application of the *inertia principle* is applied after inheritance and after the composition of events in order to allow additional effects at the level of specialization and in order to avoid inconsistency in compound events. *Class constraints* are also implicitly added to the effect clause. The *inertia principle is relaxed* for derived model elements. This keeps the class constraints consistent without the obligation for the analyst to explicitly specify possible changes to the involved derived model elements.

- *Patterns for modelling properties.* We suggested the introduction of the modelling constructs of *n-ary characteristics* and *n-ary events*. This allows the analyst to attach a characteristic or an event to more than one class. As associations, characteristics and events must be attached to *all* involved classes. We forbade the use of objects as explicit arguments in events and characteristics. We warned against the misuse of unique characteristics as explicit arguments in events. We also forbade the use of class-scoped characteristics and class-scoped events. We proposed a more neutral mechanism for implicit queries in view of possible information requirements and in view of the specifications of constraints and events. This approach solves the problem to which class a characteristic or an event must be attached in case more than one object is involved and avoids arbitrary decisions and design suggestions. The tendency to attach a characteristic or an event to a class in function of the possible user or in

function of the demanded functional requirements is also avoided. This approach also leads to a more declarative specification of events and constraints in the sense that no implementation strategy is suggested and that no arbitrary decisions about the specifications must be made. Long navigation paths in the specification are avoided. N-ary features allow making abstraction of classes and associations. In general, this approach promotes the principle of no-choice.

- *Patterns for modelling restrictions.* We proposed the introduction of the construct of a *class constraint* to represent a restriction on static properties. A class constraint must be satisfied at all times by all the objects to which it applies. Restrictions on dynamic properties must be modelled with the help of *event constraints*. Event constraints only apply to all occurrences of the event to which they apply. Additionally, the semantics of a constraint are complemented with the semantics of the *principle of non-violation*. This principle states that occurrences of events that would violate a constraint are simply rejected. In such a case, the state of the model remains unmodified. If no constraints are about to be violated, the effect of the event, which is specified in the *effect clause*, is guaranteed. Consequently, the effect clause must be written in a way that no contradictions can occur. In such a context, effect clauses are not considered as constraints. This pattern separates the specification of the business rule from the specification of the effect of an event. It also satisfies the principle of uniqueness and consequently promotes the extendibility and the adaptability of the conceptual model. Compared with the more traditional approach, which uses preconditions, postconditions and invariants, the proposed approach also avoids consistency problems, permits to describe precisely the desired semantics of restrictions at the level of analysis and promotes the principle of no-choice. Furthermore, this approach allows specializing class constraints and event constraints without the need of changing existing specifications, contra-intuitive specifications and extra classes.

- *Patterns for modelling classification*. We suggested reserving the generalization relationship construct between two classes to indicate a permanent set inclusion relationship between the subclass and the superclass. The construct of a *subset constraint* can be used to formally

specify the relationship between the superclass and the subclass. A common model element is no necessary nor sufficient condition to generalize. We also rejected the construct of an exception that enables a subclass to state that it will not inherit some specifications from its superclass. An immediate advantage of this approach is that it enables polymorphism. During analysis, we defined polymorphism as the capability to reason about a superset of objects and the capability to refine such reasoning at the level of the subclass. This approach avoids "spaghetti-generalization" in which the use and the direction of the generalization relationship would become more arbitrarily. Spaghetti-generalization would violate the principle of no-choice. Furthermore, we proposed the introduction of *objects in a class diagram* as such and as a specialization of existing classes. The advantage is that the presence of objects allows representing particulars. In *adopting a mixed scope*, some model elements are attached to specialized classes and other to more generalized classes. Adopting such a mixed scope promotes the principles of abstraction and completeness. The adoption of the mixed scope avoids explicitly modelled class constraints and it avoids more complex class constraints. It also avoids that a unique characteristic of an object is misused to identify the object.

- *Patterns for modelling functional requirements*. We proposed to divide the conceptual model into two layers: *a domain layer*, in which the real-world facts are represented, and a *functionality layer*, in which the functional requirements are specified. This approach obliges the developer to make a clear distinction between real-world facts and functional requirements. By separating the process of specifying the functional requirements from specifying the domain knowledge, the analyst is able to focus in a first phase on the real-world facts. Developers often tend to mix these specifications. This pattern forces the developer to specify the functional requirements in terms of existing specifications in the domain layer. Consequently, it forces the developer to start modelling the reality and only then go on to consider in detail the functionality of the system. As we already claimed in chapter 1, a functional requirement cannot be precisely specified without detailed knowledge of the relevant real world. Furthermore, the functionality layer can be used as a way to control the principles of completeness and abstraction. This pattern also satisfies the principle of preciseness. This pattern promotes the goal-

orientness of the specification of the functional requirements and does not focus on system interaction, which can hide the user goals.

1.3 Principles for Conceptual Modelling

We defined seven principles or quality criteria for conceptual modelling. These quality criteria can help to evaluate adequate conceptual modelling constructs and guidelines. In contrast with some quality criteria defined in the literature, these criteria are more concretely defined and they are easier verifiable. Our main contribution was in the definition of the principles of completeness, abstraction and extendibility:

- *The principle of completeness*. This principle states that *all* relevant real-world facts must be modelled. This principle is violated if a functional requirement cannot be specified because a real-world fact is not represented in the conceptual model. Such a violation is easy to verify. This principle is strongly related with the principle of abstraction.

- *Principle of abstraction*. This principle states that *only* the relevant real-world facts must be modelled. The principle of abstraction and completeness can be summarized as a directive to model *all* and *only* the relevant real-world facts. The principle of abstraction is violated if a real-world fact is represented that is not necessary to specify the functional requirements. This principle is also violated if additional real-world facts are needed to represent a given real-world fact. For instance, some methods oblige to specify a constructor for each class. Notice that this principle is very strict. A lot of guidelines and constructs lead to models that represent more facts than strictly relevant. We claim that it is an advantage when an analyst is capable to represent only the relevant facts. Additionally, it must be possible to extend the conceptual model in an easy way when other real-world facts become relevant. Therefore, the principle of abstraction is strongly related with the principle of extendibility.

- *Principle of extendibility*. This principle states that if a conceptual model must be extended, the existing specifications do not need to be changed

or removed. This principle is violated if existing specifications must be changed or removed in case of an extension of the model.

2 Future Work

As we are convinced that the knowledge about precise representation of real-world facts and functional requirements is still in its infancy, many possibilities for enhancement and further research still exist. We mention some here:

- *Adding new patterns to the analysis pattern catalog*. We do not claim that the proposed patterns cover all possible types of real-world facts and all possible conceptual modelling problems. Consequently, new patterns are needed. For instance, no pattern exists that describes how to model a property in which a certain set of objects is involved.

- *Refinement of existing patterns.* Not all proposed patterns have been worked out in full detail. Consequently, existing patterns must be further refined. For instance, how do we specify the multiplicity of more complex characteristics, which represent n-ary relations? Functional requirements can be specified in different ways in the functionality layer. How do we structure and specify these functional requirements? How do we specify a dependency between characteristics of different classes, if a long navigation path between these classes exists? Do we apply the law of Demeter [Lieb89]?

- *Patterns for transforming conceptual models into design models and programming code.* This research is situated in the software design and programming area. It is one of the key challenges in software engineering and software automation. If one is in favour of generally practised analysis and consequently of requirements that are brought as central issue into the design phase, performing analysis must result in a cost-effective added value. We believe that this cost-effective added value can only be obtained when it is possible to generate in a standardized way design models and programming code from given conceptual models.

- *Adapting the UML for conceptual model purposes.* We are in full favour of the standardization efforts made by the UML, but as described in section 1.1, the UML must be further refined to become an adequate language for conceptual modelling.

Bibliography

[Appl97] APPLETON B., *Patterns and Software: Essential Concepts and Terminology*, Object Magazine Online, Vol. 3, no. 5, 1997. Newest revision available at http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html.

[Akeh01] AKEHURST D. and BORDBAR B., *On Querying UML Data Models with OCL*, The Unified Modeling Language, Lecture Notes in Computer Science, Volume 2185, 2001, pp. 91-103.

[Asse96] ASSENOVA P. and JOHANNESSON P., *Improving Quality in Conceptual Modelling by the Use of Schema Transformations*, International Conference on Conceptual Modeling, 1996, pp. 277-291.

[Beka01] BEKAERT P., STEEGMANS E., *Non-Determinism in Conceptual Models*, 2001, Proceedings of the 10th OOPSLA Workshop on Behavioral Semantics 2001, pp. 24-34.

[Beka02] BEKAERT P. DELANOTE G. DEVOS F. and STEEGMANS E., *Specialization/Generalization in Object-Oriented Analysis: Strengthening and Multiple partitioning*, Advances in Object-Oriented Information Systems, Lecture Notes in Computer Science, Volume 2426, 2002, pp 34-43.

[Boda98] BODART F., PATEL A., SIM M. and WEBER R., *Should the optional property construct be used in conceptual modeling? A theory and three empirical tests.* Unpublished working paper. University of Queensland, Australia, 1998.

[Booc94] BOOCH G., *Object-Oriented Analysis and Design with Applications*, 1994, The Benjamin/Cummings Publishing Company.

[Booc99] BOOCH G., RUMBAUGH J., and JACOBSON I., *The Unified Modeling Language, User Guide*, 1999, Addison-Wesley.

[Bott01] BOTTONI P., KOCH M., ARISI-PRECISSI F. and TAENTZER G., *A visualization of OCL Using Collaborations*, Proceedings of the Fourth International Conference on the Unified Modeling Language, Lecture Notes in Computer Science, Volume 2185, 2001, pp. 257-271.

[Borg85] BORGIDA A., *Features of Languages for the Development of Information Systems at the Conceptual Level*, IEEE Software, Volume 2, January 1985, pp. 63-73.

[Borg88] BORGIDA A., *Modeling Class Hierarchies with Contradictions,* Proceedings of the ACM *SIGMOD* International Conference on Management of Data, 1988, pp. 434 – 443.

[Borg92] BORGIDA A., *The frame problem in object-oriented specifications: an exhibition of problems and approaches*, Technical Report LCSR-TR-209, Dept. of Computer Science, Rutgers University, New Brunswick, USA, 1992.

[Borg95] BORGIDA A., MYLOPOULOS J. and REITER R. *On the Frame problem in Procedure specifications*, IEEE Transaction on Software Engineering, Volume 21, October 1995, pp. 785 –798.

[Brac01] BRACHMAN R. and LEVESQUE H., *Knowledge Representation and Reasoning*, 2001.

[Brow02] BROWN D., *An introduction to Object-Oriented Analysis, Objects and UML in plain English*, 2002, John Wiley & Sons.

[Casa00] CASANOVA M., WALLET T. and D'HONDT M., *Ensuring Quality of Geographic Data with UML and OCL*, Proceedings of the third International Conference on the Unified Modeling Language, Lecture Notes in Computer Science, Volume 1939, 2000, pp. 225-239.

[Coad90] COAD P. and YOURDAN E. *Object-oriented analysis*, 1990, Prentice Hall.

[Cook94] COOK S. and DANIELS J., *Designing Object Systems: Let's get formal*, Journal of Object *Oriented* Programming, Volume 7, 1994, pp. 22-24.

[Cook99] COOK S., KLEPPE A., MITCHELL R. WARMER J. and WILLS A., *Defining the Context of OCL Expressions*, Proceedings of the second International Conference on the Unified Modeling Language, Lecture Notes in Computer Science, Volume 1723, 1999, pp. 372-383.

[Davi93] DAVIS A., *Software Requirements: Objects, Functions and States*, 1993, Prentice-Hall.

[Devo01] DEVOS F. and STEEGMANS E., *The Message Paradigm in Object-Oriented Analysis*, Proceedings of the fourth International Conference on the Unified Modeling Language, Lecture Notes in Computer Science, Volume 2185, 2001, pp.182-193.

[Devo03] DEVOS F. and STEEGMANS E., *Meta-Model Patterns in Object-Oriented Analysis*, Proceedings of 14th International Conference on Information Resources Management, 2003, pp. 478-480.

[Dies00] DIESTE O., JURISTO N., MORENO A., PAZOS J. and SIERRA A., *Conceptual Modelling in Software Engineering and Knowledge Engineering: Concepts, Techniques and Trends*, Handbook of Software Engineering and Knowledge Engineering, 2000, World Scientific Publishing Company.

[Dill90] DILLER A., *Z, An introduction to Formal Methods*, 1990, John Wiley & Sons.

[Dock99] DOCKX J. and STEEGMANS E., *Objectgericht Programmeren met Java*, 1999, Acco.

[Dsou99] D'SOUZA D. and WILLS A., *Objects, Components and Frameworks with UML, The Catalysis Approach*, 1999, Addison-Wesley.

[Duco02] DUCOURNAU R., *Real World as an Argument for Covariant Specialization in Programming* and *Modeling*, Advances in Object-Oriented Information Systems, Lecture Notes in Computer Science, Volume 2426, 2002, pp 3-12.

[Elma00] ELMASRI R. and NAVATHE S., *Fundamentals of Database Systems*, Addison-Wesley, 2000.

[Embl95] EMBLY D., JACKSON R. and WOODFIELD S., *OO Systems Analysis: Is It or Isn't It?*, IEEE Software, Volume 12, No. 4, 1995, pp. 19- 33.

[Erik00] ERIKSSON H. and PENKER M., *Business Modeling with UML, Business Patterns at Work*, 2000, John Wiley and Sons.

[Espe02] ESPERANZA M. and CAVERO J.M., *Hierarchies in Object Oriented Conceptual Modeling*, Advances in Object-Oriented Information Systems, Lecture Notes in Computer Science, Volume 2426, 2002, pp. 24-33.

[Ever01] EVERMANN J. and WAND Y., *Towards Ontologically based semantics for UML constructs*, Proceedings of the 20th International Conference on Conceptual Modeling, Lecture Notes in Computer Science, Volume 2224, 2001, pp. 354-367.

[Fern00] FERNANDEZ E., YUAN X., *Semantic Analysis Patterns*, Proceedings of the 19th International Conference on Conceptual Modeling, Lecture Notes of Computer Science, Volume 1920, 2000, pp. 183 – 195.

[Fiad92] FIADEIRO J., SERNADAS C., MAIBAUM T. and SERNADA C., *Describing and structuring objects for conceptual schema development*, Conceptual Modelling, Databases and CASE, 1992, pp. 117-138.

[Fowl] FOWLER M., *Analysis Patterns 2*, http:// www.martinfowler.com/ap2/index.html

[Fowl97] FOWLER M., *UML Distilled , Applying the Standard Object Modeling Language*, 1997, Addison-Wesley.

[Fowle97] FOWLER M., *Analysis Patterns, Reusable Object Models*, 1997, Addison-Wesley.

[Fran03] FRANKEL D., *Model Driven Architecture, Applying MDA to Enterprise Computing*, 2003, Wiley Publishing.

[Gamm95] GAMMA E., HELM R., JOHNSON R. and VLISSIDES J., *Design Patterns Elements of Reusable Object-Oriented Software*, 1995, Adisson-Wesely.

[Gene02] GENERO M., OLIVAS J., PIATTINI M. and ROMERO F., *Assessing Object-Oriented Conceptual Models Maintainability*, Proceedings of the 1st International Workshop on Conceptual Modeling Quality, 2002, pp. 49-60.

[Gogo98] GOGOLLA M. and RICHTERS M., *Transformation Rules for Unified Modeling Language Class Diagrams*, Proceedings of the Unified Modeling Language 98 Workshop, 1998, Lecture Notes in Computer Science, Volume 1618, pp. 92-106.

[Gogol98] GOGOLLA M. and RICHTERS M., *On Constraints and Queries in UML*, The Unified Modeling Language – Technical Aspects and Applications, Physica-Verlag, Heidelberg, 1998, pp. 109-121.

[Gree94] GREENSPAN S., MYLOPOULOS J. and BORGIDA A., *On formal Requirements Modeling* Languages*: RML revisited, International Conference on Software Engineering*, 1994, pp. 135-147.

[Guid97] GUIDE, *Defining Business Rules - What are they really?*, 1997, http://www.guide.org

[Henn02] HENNICKER R., HUSSMANN H. and BIDOIT M., *On the Precise Meaning of OCL Constraints*, Advances in Object Modeling with OCL, Lecture Notes in Computer Science, Volume 2263, 2002, pp. 69-84.

[Hoyd93] HOYDALSVIK G. and SINDRE G., *On the purpose of Object-Oriented Analysis*, In OOPSLA '93 Proceedings, 1993, pp. 240-245.

[Hubb98] HUBBERS J.W., *Helder modelleren met objecten, Moderne toepassingen vragen om OO-Modellering*, Informatie, Maart 1998, pp. 22-27.

[IBM03] IBM RESEARCH, *Multi-Dimensional Separation of Concerns*, http://www.research.ibm.com/hyperspace.

[ISO82] INTERNATIONAL STANDARD ORGANIZATION, *Concepts and Terminology for the Conceptual Schema and the Information Base*, 1982, Report Nr. 695, ISO/TC9/SC5/WG3.

[Jack83] JACKSON M., *System Development*, 1983, Prentice Hall.

[Jack95] JACKSON M., *Software Requirements and Specifications*, 1995, Addison-Wesley.

[Jack02] JACKSON M., *Some Basic Tenets of Description*, Journal on Software and System Modeling, Volume 1, September 2002, pp. 5-9.

[Jaco93] JACOBSON I., *Object-Oriented Software Engineering, a Use Case Driven Approach*, Addison-Wesley, 1993.

[Joha98] JOHANNESON P., WOHED P., *Deontic Specification Patterns Generalisation and Classification*, First International Conference on Formal Ontologies in Information Systems, 1998.

[Kent99] KENT S. and HOWSE J., *Mixing Visual and Textual Constraint Languages*, Proceedings of UML'99, 1999, IEEE Computer Society Press, pp. 384-398.

[Klas02] KLASSE OBJECTEN, *Errata for The Object Constraint Language, Precise Modeling with Unified Modeling Language*, 2002, http://www.klasse.nl/english/boeken/ocl-intro.html.

[Klep03] KLEPPE A., WARMER J. and BAST W., *MDA Explained, The Model Driven Architecture: Practice and Promise*, 2003, Addison-Wesley.

[Kost01] KOSTER G., SIX H. and WINTER M., *Coupling Use Cases and Class Models as a Means for Validation and Verification of Requirements Specifications*, Requirements Engineering, Volume 6, 2001, pp.3-17.

[Larm98] LARMAN C., *Applying UML and Patterns, An introduction in object-oriented analysis and design*, 1998, Prentice Hall.

[Lieb89] LIEBERHERR K. and HOLLAND I., *Assuring Good Style for Object-Oriented Programs*, IEEE Software, September 1989, pp. 38-47.

[Lind94] LINDLAND O., SINDRE G. and SOLVBERG A., *Understanding Quality in Conceptual Modeling*, IEEE Software, Volume 11, No. 2, 1994, pp. 42-49.

[Lisk94] LISKOV B. and WING J., *A Behavioral Notion of Subtyping*, ACM Transactions on Programming Languages and Systems, Volume 6, November 1994, pp.1811-1841.

[Macg92] MACGINNES S., *How objective is Object-Oriented Analysis?*, Conference on Advanced Information Systems Engineering, CaiSE 1992, Lecture Notes in Computer Science, Volume 593, pp. 1-16.

[Macl82] MACLENNAN B., *Values and Objects in Programming Languages*, SIGPLAN Notices, Volume 17, Number 12, pp.70-79, 1982.

[Mala01] MALAN R. and BREDEMEYER D., *Functional Requirements and Use Cases*, White paper, 2001, Bredemeyer Consulting.

[Meye97] MEYER B., *Object-Oriented Software Construction*, 1997, Prentice Hall.

[Miss98] MISSIAEN L. and MARTENS B., *Temporal Logics and Temporal Reasoning*, Department of Computer Science, K.U. Leuven, 1998.

[Moss02] MOSSE F., *Modeling Roles, A practical series of Analysis Patterns*, Journal of Object Technology, Volume 1, No 4, pp. 27-37, http://www.jot.fm.

[Ngu89] NGU A., *Conceptual Transaction Modeling*, IEEE Transactions on Knowledge and Data Engineering, Volume 1, No. 4, December 1989, pp. 508 - 518.

[OMG01] OBJECT MANAGEMENT GROUP, *Unified Modeling Language Specification*, Version 1.4, September 2001, http://www.omg.org/technology/documents/formal/uml.htm

[Opda02] OPDAHL A. and HENDERSON-SELLERS B., *Ontological Evaluation of the UML Using the Bunge-Wand-Weber Model*, Journal on Software and System Modeling, Volume 1, September 2002, pp. 43-67.

[Past01] PASTOR O., GOMEZ J., INSFRAN E. and PELECHANO V., *The OO-Method approach for information systems modeling: from object-oriented modeling to automated programming*, Information Systems, Volume 26, 2001, pp. 507-534.

[Peet01] PEETERS F., VONKEN F., *Objectgeoriënteerde Domeinanalyse*, 2001, Academic Service.

[PLoP] Pattern Languages of Programs, http://jerry.cs.uiuc.edu/~plop/

[Poel98] POELS G., *An analytical Evaluation of Static Coupling Measures for Domain Object Classes*, ECOOP Workshops, 1998, pp.260-263.

[Rich99] RICHTERS M. and GOGOLLA M., *A Metamodel for OCL*, Advances in Object Modelling with OCL, Lecture Notes in Computer Science, Volume 1723, 1999, pp. 156-171.

[Rich02] RICHTERS M. and GOGOLLA M., *OCL: Syntax, Semantics and Tools*, Advances in Object Modeling with OCL, Lecture Notes in Computer Science, Volume 2263, 2002, pp. 42-68.

[Roll92] ROLLAND C. and CAUVET C., *Trends and Perspectives in Conceptual Modelling*, Conceptual Modelling, Databases and CASE: an Integrated View of Information Systems Development, 1992, John Wiley.

[Rumb99] RUMBAUGH J., JACOBSON I. and BOOCH G., *The Unified Modeling Language Reference Manual*, 1999, Addison-Wesley.

[Said03] SAID J., *Pattern-Based Approach for Object-Oriented Software Design*, PhD Thesis, 2003, Departement of Computer Science, K.U.Leuven.

[Schr91] SCHREFL M. and KAPPEL G., *Cooperation Contracts*, Proceedings of the International Conference on Conceptual Modeling, 1991, pp.285-307.

[Schu98] SCHUETTE R. and ROTTHOWE T., *The Guidelines of Modeling – An approach to Enhance the Quality in Information Models*, Proceedings of the 17th International Conference on Conceptual Modeling, 1998, Lecture Notes of Computer Science, Volume 1507, pp. 240 – 254.

[Send00] SENDALL S. and STROHMEIER A., *From Use Cases to System Operation Specifications*, Proceedings of the third International Conference on the Unified Modeling Language, Lecture Notes in Computer Science, Volume 1939, 2000, pp. 1-15.

[Senda00] SENDALL S. and STROHMEIER A., *Enhancing OCL for Specifying Pre- and Postconditions,* Third International Conference on the he Unified Modeling Language, Workshop The Future of the UML Object Constraint Language, 2000.

[Shla88] SHAELR S. and MELLOR S. *Object-Oriented Systems Analysis, Modeling the World in Data*, 1988, Prentice Hall.

[Sim00] SIM E., FORGIONNE G. and NAG B., *An Experimental Investigation into the Effectiveness of OOA for Specifying Requirements,* Requirements Engineering, 2000, pp. 199-207.

[Sisa02] SI-SAID CHERFI S., AKOKA J. and COMYN-WATTIAU I., *Conceptual Modeling Quality- From EER to UML Schemas Evaluation,* Proceedings of the 21st International Conference On Conceptual Modeling, Lecture Notes of Computer Science, Volume 2503, 2002, pp. 414-428.

[Snoe99] SNOECK M., DEDENE G., VERHELST M. and DEPUYDT A., *Object-Oriented Enterprise Modelling with MERODE,* 1999, Universitaire Pers Leuven.

[Somm92] SOMMERVILLE I., *Software Engineering*, 1992, Addison-Wesley.

[Stee99] STEEGMANS E. and AL-AHMED W., *Object-oriented Programming Languages, Concepts and Application*, 1999, K.U.Leuven.

[Stee00] STEEGMANS E., BEKAERT P., DEVOS F., DOCKX J., SWENNEN B. and VAN BAELEN S., *Object-oriented Analysis with EROOS*, 2000, Department of Computer Science, K.U.Leuven.

[Stee02] STEEGMANS E. and DOCKX J., *Objectgericht programmeren met JAVA*, 2002, Acco.

[Stev02] STEVENS P., *On the interpretation of binary associations in the Unified Modelling Language*, Journal on Software and System Modeling, Volume 1, Number 1, September 2002, pp. 68-79.

[Vanb93] VAN BAELEN S., LEWI, J., STEEGMANS, E. and SWENNEN B., *Constraints in Object-Oriented Analysis*, Object Technologies for Advanced Software, Lecture Notes in Computer Science, Volume 742, 1993, pp. 393-407.

[Vanb94] VAN BAELEN S., LEWI J., STEEGMANS E., *Constraints in Object-Oriented Analysis and Design*, Technology of Object-Oriented Languages and Systems, TOOLS 13, Prentice-Hall, 1994, pp. 185-199

[Vane01] VAN ECK P., ENGELFRIET J., FENSEL D., VAN HARMELEN F., VENEMA Y. and WILLEMS M., *A Survey of Languages for Specifying Dynamics: A Knowledge Engineering Perspective*, IEEE Transaction on Knowledge and Data Engineering, Vol. 13, No. 3, 2001, pp. 462-496.

[Wand93] WAND Y. and WEBER R., *On the ontological expressiveness of information systems analysis and design grammars.* Journal of Information Systems, 1993, pp. 217-237.

[Warm99] WARMER J. and KLEPPE A., *The Object Constraint Language, Precise modeling with UML*, 1999, Addison-Wesley.

[Warm03] WARMER J. and KLEPPE A., *The Object Constraint Language, Getting Your Models Ready for MDA*, 2003, Addison-Wesley.

[Wier91] WIERINGA R.J., WIEGAND H., MEYER J., DIGNUM F.P.M., *The inheritance of Dynamic and Deontic Integrity Constraints*, Annals of Mathematics and Artificial Intelligence, Volume 3, No. 2-4, 1991, pp. 393-428.

[Zeic02] ZEICHICK A., *Modeling Usage Low; Developers Confused About UML 2.0, MDA*, Software Development Times, July 2002, http://www.sdtimes.com/ news/058/story3.htm.