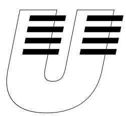


UNIVERSITÄT
KOBLENZ · LANDAU

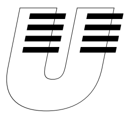
Institut für
Wirtschaftsinformatik
Fachbereich Informatik
Universität Koblenz-Landau

pULRICH FRANK

THE MEMO OBJECT MODELLING LANGUAGE (MEMO-OML)

Juni 1998

Arbeitsberichte des Instituts für Wirtschaftsinformatik Nr. 10



ULRICH FRANK

THE MEMO OBJECT MODELLING LANGUAGE (MEMO-OML)

Juni 1998

Arbeitsberichte des Instituts für Wirtschaftsinformatik Nr. 10

Die Arbeitsberichte des Instituts für Wirtschaftsinformatik dienen der Darstellung vorläufiger Ergebnisse, die i.d.R. noch für spätere Veröffentlichungen überarbeitet werden. Die Autoren sind deshalb für kritische Hinweise dankbar.

The "Arbeitsberichte des Instituts für Wirtschaftsinformatik" comprise preliminary results which will usually be revised for subsequent publications. Critical comments would be appreciated by the authors.

Alle Rechte vorbehalten. Insbesondere die der Übersetzung, des Nachdruckes, des Vortrags, der Entnahme von Abbildungen und Tabellen - auch bei nur auszugsweiser Verwertung.

All rights reserved. No part of this report may be reproduced by any means, or translated.

Anschrift des Verfassers/
Address of the author:

Prof. Dr. Ulrich Frank
Institut für Wirtschaftsinformatik
Universität Koblenz-Landau
Rheinau 1
D-56075 Koblenz

Arbeitsberichte des Instituts für Wirtschaftsinformatik
Herausgegeben von / Edited by:

Prof. Dr. Ulrich Frank
Prof. Dr. J. Felix Hampe

©IWI 1998

Bezugsquelle / Source of Supply:

Institut für Wirtschaftsinformatik
Universität Koblenz-Landau
Rheinau 1
56075 Koblenz

Tel.: 0261-9119-480
Fax: 0261-9119-487
Email: iwi@uni-koblenz.de
WWW: http://www.uni-koblenz.de/~iwi

Contents

Abstract 5
1. Introduction 6
2. The MEMO Meta-Metamodel 6
2.1 The Notation 8
2.2 Relationship to other MEMO Languages
3. Requirements 11
4. Language Features 14
4.1 Basic Concepts (Information Analysis) 15
4.1.1 Attributes and Services. 15
4.1.2 Associations 15
4.2 Specialisation 16
4.3 Additional Concepts (Object Design) 26
4.3.1 Attributes/Associations 26
4.3.2 Services 27
4.3.3 Guards and Triggers 28
4.3.4 Constraints 28
4.3.5 Subtyping (Refinements), Interfaces, Metaclass 28
4.3.6 Organisation of Object Models 30
4.4 Implementation-Oriented Refinements (System Design) 31
4.4.1 Stored References. 31
4.4.2 Persistence 31
4.4.3 User Interface Concepts 32
5. The Metamodel 32
5.1 Basic Concepts 33
5.2 Associations 43
5.3 Metaclasses 50
5.4 Organisation Concepts 50
5.5 Stored References to Associated Objects. 52
5.6 Persistence 53
5.7 User Interface Concepts 53
5.8 The Graphical Notation 55
5.8.1 Naming Conventions 55
5.8.2 Graphical Symbols 59
6. User Interface Classes 64
7. Future Work 65
References 66

Abstract

"Multi Perspective Enterprise Modelling" (MEMO) is a method to support the development of enterprise models. It suggest a number of abstractions which allow to analyse and design various interrelated aspects like corporate strategy, business processes, organizational structure and information models. Any of those views can be modelled with a specific modelling language or diagram technique respectively. In order to allow for a tight integration of the various perspectives, the modelling languages suggested by MEMO are based on common concepts. Those concepts are defined within a common meta-metamodel.

Each of the languages within the MEMO framework can be regarded as a specialised language/terminology to serve the purpose of specific tasks. One of those tasks is the design of information systems. For this purpose MEMO suggests an object-oriented approach. The corresponding language, MEMO-OML (MEMO Object Modelling Language), is intented to support analysts and designers of information systems. In the current version, it is mainly restricted to the description of static aspects. MEMO-OML provides concepts which are suited to cover a large part of the software lifecycle. The concepts are specified in a semi-formal way with a graphical metamodel. It defines the abstract syntax and semantics of MEMO-OML. Additionally, a graphical notation is introduced. The metamodel is supplemented by comprehensive discussions of various language features.

1. Introduction

A modelling language is an instrument which should be designed to fit its purpose. Within the MEMO ("Multi Perspective Enterprise Modelling") method we focus on models that support the analysis and design of *corporate information systems*. This is usually a complex endeavour that is not restricted to designing, implementing and integrating software. Instead, it recommends to analyse and (re-) design a company's organisation and may be its strategy. In order to support these different views on the enterprise, MEMO offers a variety of specialised languages, for instance a language to model business processes, the organisational structure or the corporate strategy. Within these languages the MEMO-OML is of outstanding importance. This is for two reasons. Firstly, it is used in almost any stage of enterprise modelling. Secondly, it serves to reconstruct the other modelling languages in order to use them within a tool (see fig. 4).

Due to the nature of corporate information systems, we consider the design of object models as the pivotal task. Therefore, the current version of MEMO-OML only allows to design object models. This is different from other object-oriented modelling languages which cover various kinds of diagrams (such as OML [FiHe96], or UML [Rat97c]). While this does not exclude, to support additional kinds of diagrams, like message flow diagrams, state transition diagrams in a later stage, it has to be taken into account that MEMO-OML is part of a method for enterprise modelling. Therefore, there is more emphasis on additional domain specific aspects - like a company's organisation or its strategy which are covered by additional modelling languages. The most important of those is MEMO-OrgML (Organisation Modelling Language) which supports the design of business process models that are integrated with a corresponding object model (for an overview see [Fra97]). There is no doubt that the value of a language depends on the extent of its use. For this reason, it makes certainly sense to standardize modelling languages. Currently the OMG is evaluating a proposal which is backed by an industrial consortium. There is evidence that this proposal, called "Unified Modelling Language" (UML, [Rat97c]), will be standardized by the OMG eventually - although the current version can be expected to be still revised a few times. Against this background we had to decide whether to adopt UML or to define our own object modelling language to be used within MEMO. Although we appreciate the benefits of standardization, we decided not to adopt UML. This is mainly for three reasons. First, the development of UML has not finished yet. We do not want to depend on a refinement process we cannot influence and which, at the same time, has a crucial impact on our concepts. Second, the UML versions we have seen so far did not convince us that UML will ever be the language of our choice. This is certainly true for the purpose of teaching, since UML comes with a vast amount of concepts which are, in part, redundant. Furthermore, some of the concepts do not seem to be as elaborated as they should be (for an evaluation of the UML see [FrPr97], 3.2). Third, we do not assume that the state of the art in modelling languages is mature enough to allow for freezing a particular proposal (see [Fra98a]). Instead, we would like to refine the languages we use with our (hopefully) progressing knowledge of the subject.

2. The MEMO Meta-Metamodel

A modelling language can be specified in various ways - like using a grammar, a metamodel, or a natural language description. Within MEMO we decided for the metamodelling approach, since it does not require a paradigm shift between object and meta level. Also, a metamodel

provides a good foundation for the implementation of modelling tools. It is an essential goal of MEMO to allow for a tight integration of various models within an overall enterprise model. In order to support the integration of the modelling languages offered by MEMO, they are all specified by concepts which are defined in a common meta-metamodel (see fig. 1). For a detailled description of the meta-metamodel and its comparison to alternative meta-metamodels see ([Fra98b]).

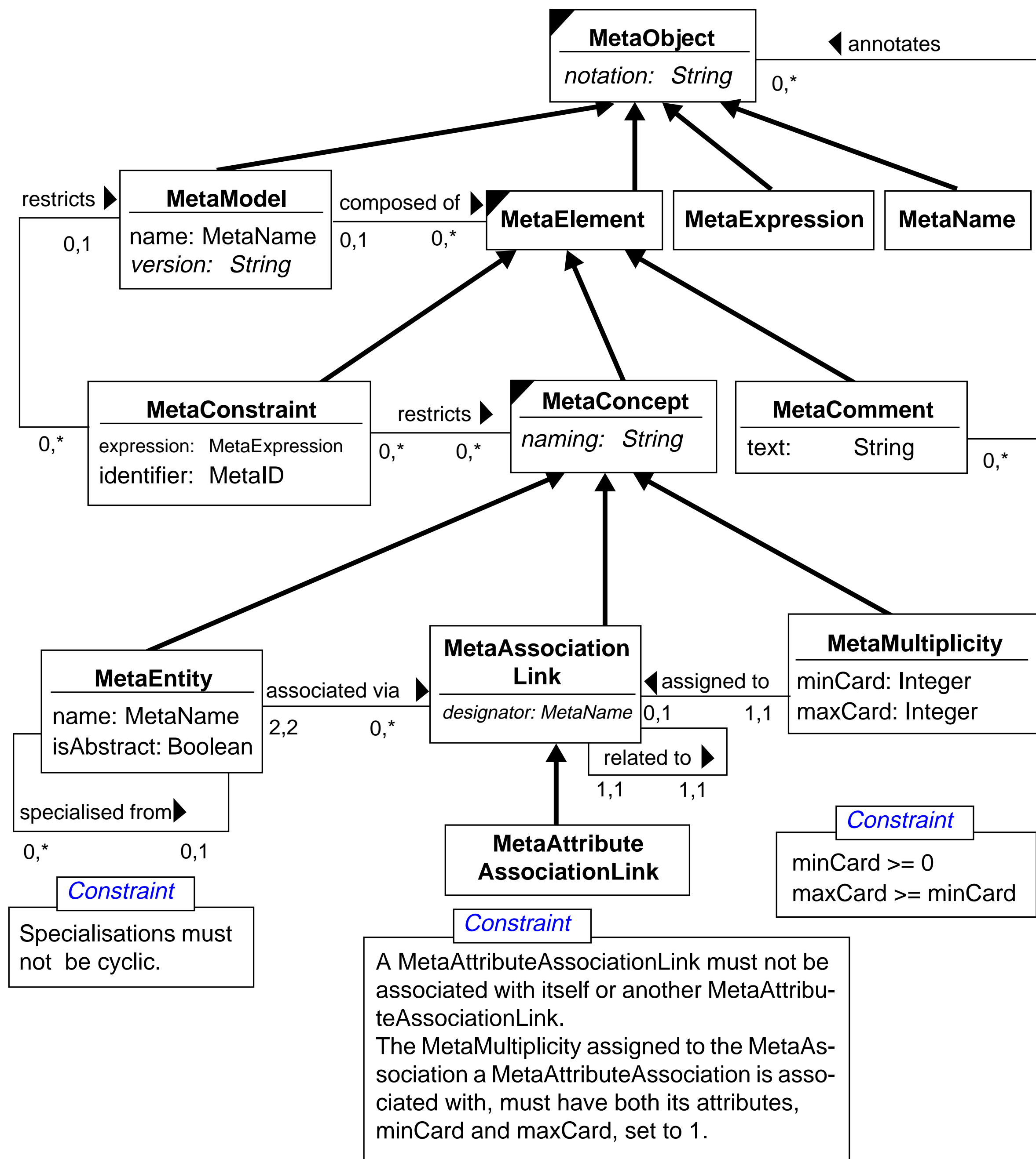


Fig. 1: The MEMO-OML Meta-Metamodel

2.1 The Notation

The notation used for designing metamodels is much like the one already used for the meta-metamodel, except for a few additional symbols. In order to give a more precise, though not intended to be formal, specification of the notation, we will differentiate between the graphical representation and the textual designators/annotations. For the latter we use a Bachus-Naur form. The bold faced non-terminal symbols are used within the graphical illustration of the notation (see fig. 2 and fig. 3). Notice that we do not bother with specifying a few basic non-terminal symbols - like *String*, *LowercaseLetter*, *UppercaseLetter*, *LineFeed* etc.

Basic Symbols

<digit> ::= 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
<positiveInteger> ::= {< digit >}
<infiniteNumber> ::= '*'
<separator> ::= ', '
<lowerString> ::= <LowercaseLetter> <String>
<upperString> ::= <UppercaseLetter> <String>

Multiplicity

<maxCardinality> ::= <PositiveInteger> | <infiniteNumber>
<minCardinality> ::= <PositiveInteger>
<multiplicity> ::= '(' <minCardinality> separator <maxCardinality> ')'

MetaEntity

<entityName> ::= <upperString>

Attribute

<attributeName> ::= <lowerString>

Constraint

<constraintkey> ::= 'C' <number>
<MetaExpression> ::= <OCLExpression> | <GRALExpression> | <String>

Association

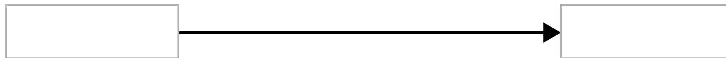
<backwardArrow> ::= '◀ '
<forwardArrow> ::= '▶ '
<designator> ::= <lowerString>
<backwardDesignator> ::= <backwardArrow> <designator>
<forwardDesignator> ::= <designator> <forwardArrow>
<forwardFirst> ::= <forwardDesignator> [< LineFeed> <backwardDesignator>]
<backwardFirst> ::= <backwardDesignator> [< LineFeed> <forwardDesignator>]
<assocDesignator> ::= forwardFirst | backwardFirst

The graphical symbols used to represent the concepts defined in the meta-metamodel are rendered in fig. 2 and fig. 3.

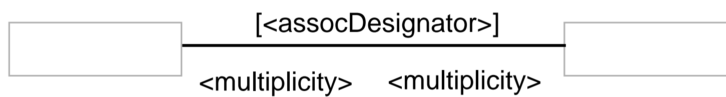
<entityName> **instance of MetaEntity**

<entityName> **abstract instance of MetaEntity**

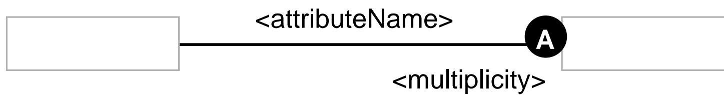
<entityName> **instance of an instance of MetaEntity**
(not permitted for abstract instances)



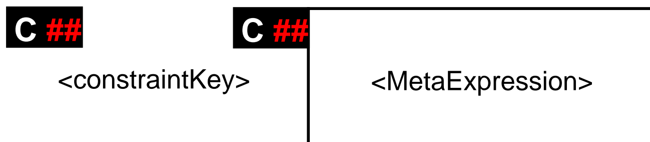
generalisation
instance of "is subclass of"-association



association
(two linked instances of MetaAssociationLink)

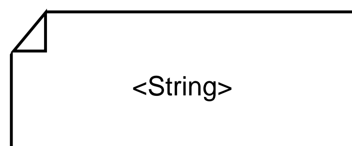


attribute
(an instance of MetaAttributeAssociationLink linked with an instance of MetaAssociationLink)



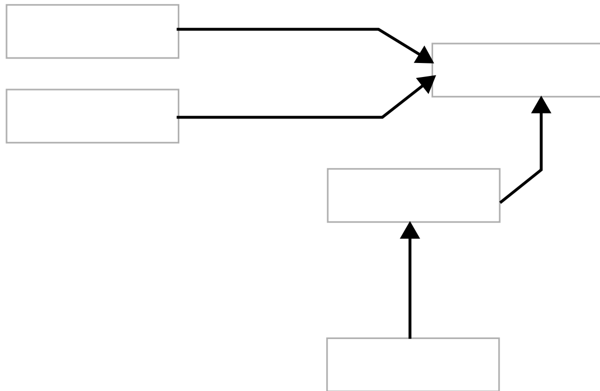
constraint
(instance of MetaConstraint, n serves as a reference)

As specified in the meta-meta-model, n is a positive integer and has to be a unique key within a model

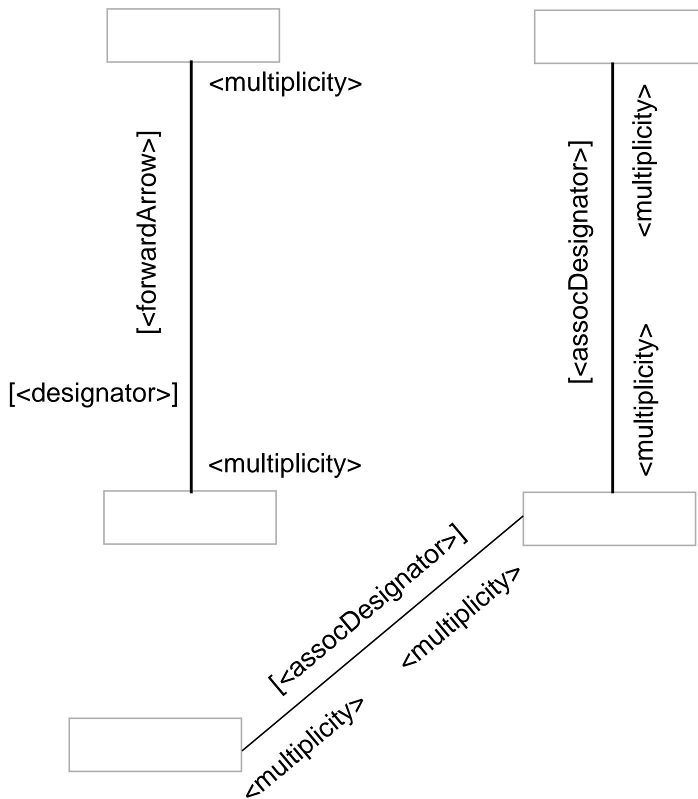


comment
(instance of MetaComment)

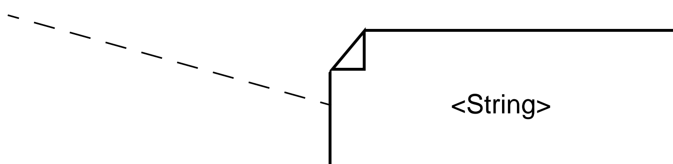
Fig. 2: Notation of MEMO Metalanguages (1): Basic Symbols



generalisation
In principle, generalisation can be represented by any line with an arrow that points to the general entity.



association
The rule that applies for the representation of associations is similar to that for generalisation: In principle, any line that connects two entities is suitable. It is recommended, but not mandatory, to render the designators and multiplicities in parallel to a line.



comment
As an option, a comment can be assigned to a specific part of a model by a dotted line.

Fig. 3: Notation of MEMO Metalanguages (2): Additional Symbols and Illustration of Alternative Representations

2.2 Relationship to other MEMO Languages

While any of the specialised modelling languages can be used on its own in a straightforward way (e.g. using a sheet of paper and a pencil), it is desirable to use specialised modelling tools in the long run. For this purpose MEMO-OML is of essential importance: It serves to transform the various metamodels (including its own) into object models which are used to implement modelling tools (see fig. 2). Notice that this is not only a translation, but rather a reconstruction, since modelling tools require information (e.g. about users, versions) that are of no relevance for the specification of a language.

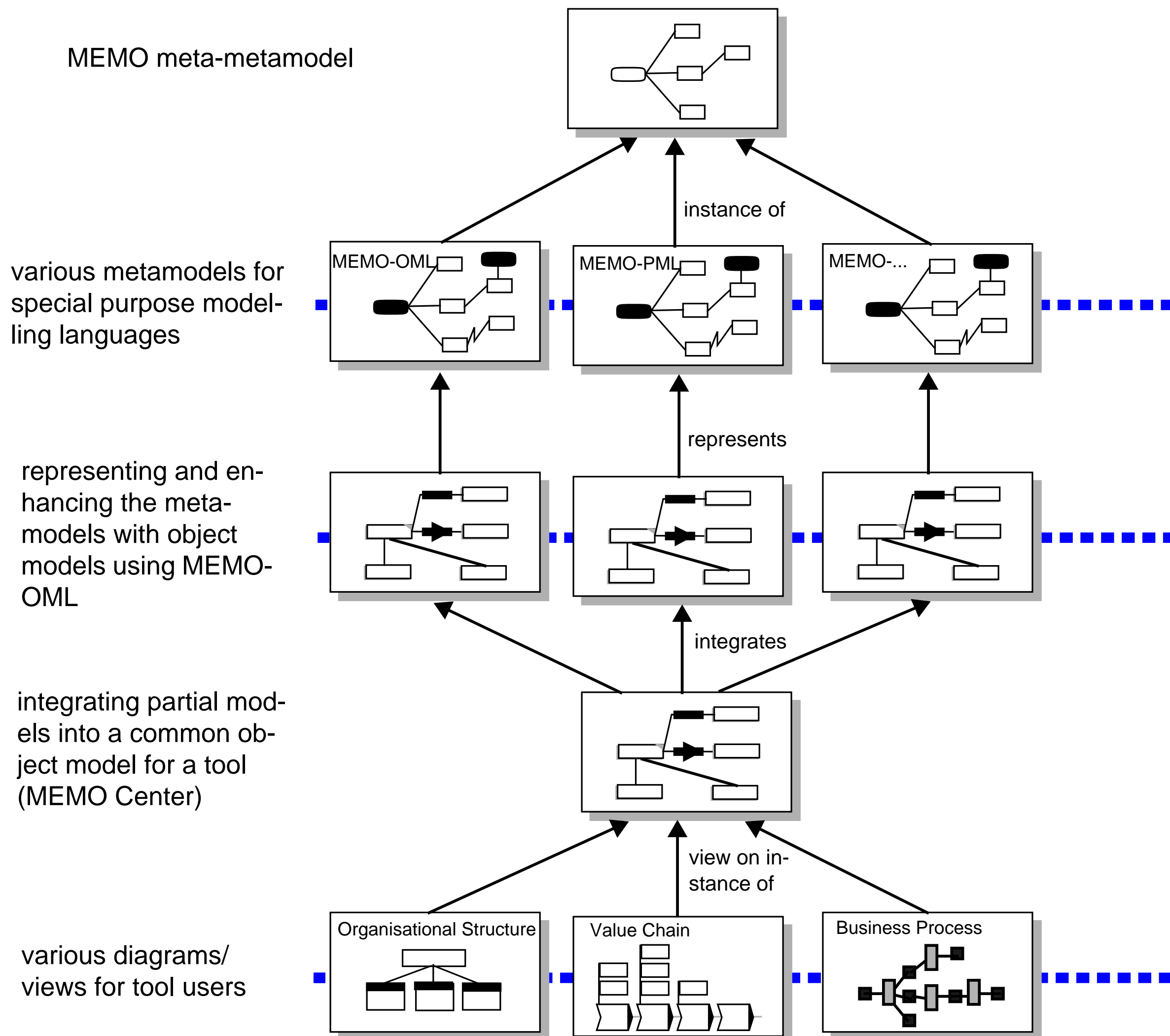


Fig. 4: Meta-Metamodel for Modelling Languages and its Relationship to Object Models for Tools

3. Requirements

In the current version, MEMO-OML focuses mainly on static aspects of object models. An object model should provide a suitable foundation for software development. At the same time, it should serve as an instrument for describing real world domains in an natural way. Both

kinds of requirements are stressing different aspects of a language which do not have to be in line. Common formal requirements to be met by software, like correctness, reliability, efficiency etc. suggest modelling languages with a high degree of formal rigour. Additionally, their concepts should correspond to concepts which are used in implementation (usually: programming) languages. On the other hand, conceptual models should serve as a medium to communicate with the various participants involved in a system development project. Therefore, they should offer understandable - not to say intuitive - abstractions also for those who do not have a background in software engineering. In other words: A modelling language should include concepts which correspond to the perceptions and conceptualisations of those who use it.

Although the discipline of software engineering does not offer a mature theory about the process of software development and the abstractions to be used within this process, there is a substantial amount of knowledge about relevant requirements and corresponding measures. This is very different with the knowledge we have about the way humans perceive modelling languages, how a language effects their ability to express and understand appropriate models. There are only a few empirical studies on how people perceive data models (for instance: [Hit95], [GoSt90]). They suggest that many people do not regard abstractions like entity relationship models as intuitive. For this reason, the concepts we suggest are based on assumptions which are influenced - or should we say: biased? - by our own experiences and preferences. It is often stated that object-orientation provides a natural way to conceputalise the world. While we could easily agree with this assumption as far as it concerns our own perception, we do not believe that it is convincing in the end. This is at least for two reasons. Firstly, to most people a natural language should allow for more intuitive descriptions. Secondly, there is no doubt that the way how people perceive and evaluate language concepts varies with their personal background. Nevertheless, there is a good reason to opt for an object-oriented approach. Compared to traditional approaches to conceptual modelling, like ER modelling, object-oriented modelling allows for a much higher level of *abstraction* - by providing concepts as information hiding or by specifying attributes with user defined classes. A higher level of abstraction not only fosters the chance to use an abstraction that is appropriate (we could also say "natural") in a certain modelling context. Furthermore, it allows to delay implementation decisions and improves the maintainability of an information system.

Considering the complexity of enterprise modelling, it is hardly possible to recommend a specific way to proceed and specific abstractions to be used. However, in order to define appropriate abstractions, we differentiate hypothetically a set of prototypical tasks which have to be performed during the development of corporate information systems (see fig. 3). Notice that we do not intend to prescribe a specific way to proceed. The configuration of a particular development process depends on various aspects: the scope of the system to be developed, the skills of the participating analysts and domain experts, the extent and qualitiy of documents available from the past, etc. This is also the case for the extent of a process. For instance: sometimes, a strategic or even an organisational analysis may be regarded as too time consuming. Other projects may focus on business process redesign only. For this reason, the process model presented in fig. 1 serves mainly to identify plausible (not: mandatory) tasks. We will then use assumptions about these tasks - concerning participants, results to be produced etc. - to outline a level of abstraction that seems to be appropriate for corresponding object models. It is one of the main advantages offered by an object-oriented approach to avoid friction between the various stages of system development. This can be basically accomplished by providing the same core concepts - namely classes and objects - for all tasks to be performed. In order to support

the various activities in an efficient way, those concepts have to be available on different levels of abstraction and formalization. Hence, a modelling language should provide coherent concepts with various degrees of detail and formal precision.

According to the prototypical process model presented in fig. 1, we will differentiate three development stages of object models: analysis models, object design models, and system design models. Again: We do not intend to enforce a particular way to proceed or particular languages to be used/not used within a development stage.

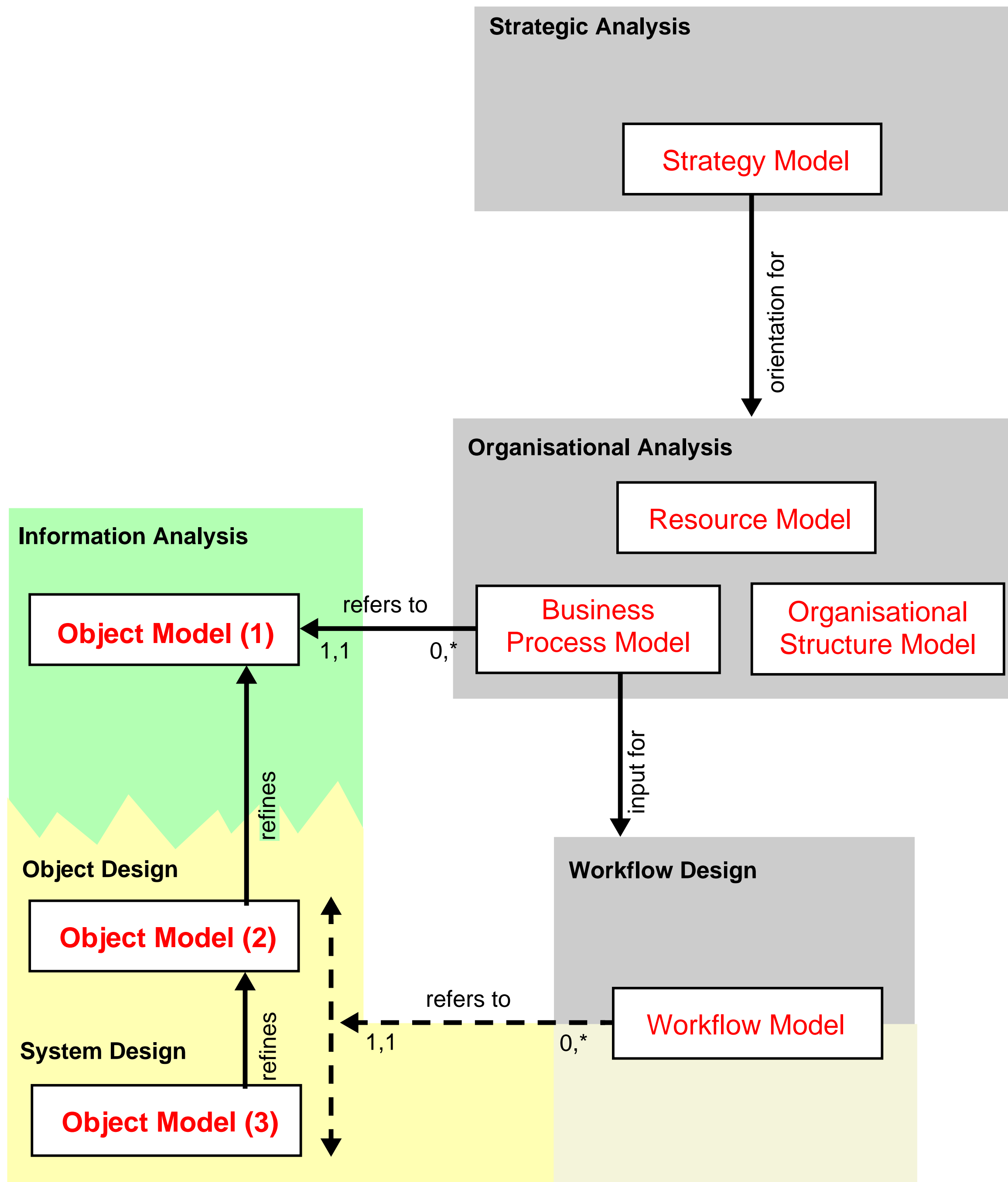


Fig. 5: Occurrences of Object Models in different Stages of System Development

Information Analysis

Within this report, our focus is on object models. Therefore, we abstract from the pecularities one has to deal with during strategic as well as during organisational analysis and design. Instead, we assume that organisational design has - among other things - resulted in a set of preliminary business process models. To further refine these models, any activity within a business process can be assigned a set of information that it requires or that it produces. The part of that information which is supposed to reside within the computerized information system to be designed serves as an initial input for building an analysis object model (object model (1) in fig. 1). This stage of analysis aims at gaining a deep understanding of the tasks to be performed in the domain of interest. Therefore, it requires intensive communication with domain experts and prospective users. For this reason, the concepts used within this stage should not be loaded with details which are only relevant for design or even implementation purposes. Among other things, this implies not to enforce a high degree of formalisation. Nevertheless, an analysis object model should allow for a seamless transformation into more detailled and formalized object models.

Object Design

Object design aims at specifying the objects identified during analysis in more detail and with more formal rigour.[1] This requires additional concepts. The participants in this stage of the life cycle are expected to be familiar with software engineering. For this reason, the concepts introduced specifically for this stage do not have to be intuitive for every domain expert. Nevertheless, they should directly correspond to concepts used during analyis in order to avoid friction. Compared to analysis there is more emphasis on the specification of services and additional constraints.

System Design

System design aims at preparing an object model for implementation. It includes further refinements of the object model and the definition of the overall system architecture. The latter recommends to have concepts available that allow to model existing software as well as base systems, such as hardware and operating systems.

The following description of MEMO-OML consists of three sections. Section 4 provides an natural language introduction to language concepts. It starts with concepts intended to be used within information analysis. Subsequently, additional concepts for object design and system design will be introduced. Against this background the metamodel (abstract syntax and semantics), as well as the graphical notation (concrete syntax), will be presented in section 5.

4. Language Features

While the differentation into information analysis, object design and system design serves to illustrate a possible use of language concepts during the various stages of system development, there is an additional, orthogonal differentiation into three categories which denote the purpose a concept is to serve. *Semantic* concepts serve to describe those properties of an object which correspond to properties of real world entities or concepts. *Organisation* concepts serve to organize/structure object models in order to foster understanding and maintainability. *Manage-*

1. Notice that this is different from the use of the term within OMT ([Rum91]).

ment concepts provide information that helps with the management of instantiated object models, like default values or a history flag for attributes (which, when it is set, indicates that the changes of the state of an attribute should be recorded). Ressource concepts allow to describe information system ressources, like hardware, operating systems, existing applications etc. that may be referenced in an object model.

4.1 Basic Concepts (Information Analysis)

An object model consists of *classes*. An *object* is an instance of exactly one class. Notice that this is different from classless programming languages and from some languages used within DBMS, such as the ODL suggested by the ODMG ([CaBa97]). At the same time it is more specific than the UML where an object may be an instance of one *or* many classes ([Rat97e]). For the description of analysis object models MEMO-OML offers a rudimentary concept of a class. A class has a name that must be unique within the scope of an object model. At this level of abstraction, the main emphasis is on semantic concepts. There is only one management concept that can be optionally assigned to a class at this stage: *Multiplicity* serves to express the minimum and maximum number of instances (*cardinalities*) of a class. The essential semantic concept to specify a class within analysis object models are *attributes* and *services*.

4.1.1 Attributes and Services

An attribute represents an object or a set (or an ordered collection) of objects of a specific class respectively. During analysis, an attribute is assumed to be specified by two concepts: *Multiplicity* and *class*. Multiplicity allows to express the minimum and maximum number of instances which together represent a particular attribute. Class serves to describe the class a particular instance of an attribute is instantiated from. A service is characterized by a *signature* which consists of its name and none to many parameters. Each parameter has to be assigned a name that is unique within a signature. Additional constraints may be defined in order to enforce certain syntactic conventions, like - for instance - those imposed by Smalltalk or Eiffel (see 5.8.1). Each service returns zero or one value. We regard parameters and returned values as objects, i.e. they are specified by classes. The object design phase requires more elaborated specifications of attributes and services. Therefore both concepts, attribute and service, will be specified later.

4.1.2 Associations

Additionally, it is possible to define associations between classes. We only allow for *binary* associations. Each class that participates in an association can be assigned a multiplicity - notice that this is different from the multiplicity assigned directly to a class. In case the maximum cardinality is larger than one, an additional predicate can be assigned (like "ordered", "sorted"). An association can be annotated with a directed designator in order to support an intuitive understanding of a model. The classes involved in an interaction assocation may be assigned a *context role* that serves to provide additional information without formal semantics. A specialised class inherits all the properties of its superclasses (except those which have been excluded because of naming conflicts). In addition to mere interaction associations, MEMO-OML offers two special types of associations: *aggregation* and *delegation*. Delegation will be described later, after introducing a first specification of specialisation.

The idea of composing parts to aggregated entities is an important concept for describing and analyzing the real world. It can be mapped directly to an object model, since both, parts and

aggregates can be represented as objects. For this reason, it makes sense to provide the concept of aggregation with an object-oriented modelling language. At the same time, we did not succeed in defining a concept of aggregation which would allow for a clear distinction from interaction associations. This is for a good reason: The intuitive semantics of aggregation is based on the perception and interpretation of real world circumstances. Those are, however, outside the scope of the formal concepts used within a model. For this reason, the formal concept of aggregation is characterized by two modest constraints:

#1 An aggregation is a *directed* association with a clear distinction between part and aggregate.

#2 Aggregations must not be cyclic.This does not exclude recursive associations. It simply means that, within an aggregation, an object must not act as a part of itself. Applying this constraints requires to take into account that aggregations are transitive.

4.2 Specialisation

While specialisation, often illustrated with the designator "is a", seems to be a common and intuitive concept for describing the world, a thorough analysis reveals that a specification of specialisation has to deal with complex challenges. These challenges are mainly caused by the fact that the concept of inheritance as it is provided by most object-oriented programming languages differs in a subtle way from the common concept of specialisation. Furthermore, within natural language specialisation is often used in an ambiguous way - which adds to its versatility but hinders a proper formal reconstruction.

The common sense concept of specialisation implies that any instance of a class is an instance of its superclass as well. Nevertheless, MEMO-OML uses a concept of specialisation where an object can only be instance of exactly one class. This restriction is hardly to avoid in order to allow for a seamless transformation into an implementation model: Object-oriented programming languages usually do not allow an object to be an instance of more than one class. At the same time, it is a restriction that imposes severe problems as will be shown later. At first sight, the specialisation of a class may be regarded as a new class that inherits all the features of this class and adds a number of features (which do not contradict the inherited features). Such a restrictive rule, however, would not allow to represent the common notion of specialisation within an object model to a satisfactory extent: It is common sense to use general propositions also in cases where exceptions exist: "A bird can fly ...". Therefore, it seems to be a good idea to allow for *redefinition* of inherited features. However, despite its versatility, redefining inherited features is similar to the use of a general proposition which does not hold in any particular case. This means, in a rigorous sense: It is wrong.

It is not acceptable to provide for arbitrary redefintions of any inherited feature (as it is possible in some object-oriented programming languages), since that would allow to completely disguise the concept of specialisation: A "specialised" class would not have to have any feature in common with its superclass. This is the reason, too, why MEMO-OML does not allow to delete inherited features (or associated objects) from a class. While we agree with Meyer to a great extent that "even with a careful design some taxonomy exceptions may remain " ([Mey97], p. 843), we do not think that arbitrary "descendant hiding", as Meyer calls it, is necessary. Notice that that does not imply to completely give up the versatility provided by deleting inherited features. By using zero as a minimum cardinality, there is a chance to avoid the use of an inherited feature (on the instance level). Nevertheless, in order to foster the construction of "natural", "intuitive" models, it is desirable to allow for certain redefinitions. Therefore

we should look for an acceptable compromise: How could the redefinition (or overriding) of inherited features be restricted in order to specify a satisfactory compromise?

There are two main perspectives to look at the problem of redefining the class of inherited features: a programming language perspective and an conceptual modelling perspective. In the area of programming languages there have been numerous investigations (for instance: [Car87], [Cas95], [Mey97], [SzOm93]) - mainly related to type checking problems. While conceptual modelling cannot completely neglect the pecularities of implementation languages, it is also directed towards the representation of "natural", common sense concepts. With a similar intention in mind, researchers in Artificial Intelligence, especially in the area of knowledge representation, have tried to model common human "communication conventions" ([MCa86], p. 91). Efforts to reconstruct human thinking with machines resulted in a number of approaches which allow exceptions to be handled in formal systems without the disastrous consequences contradictions usually produce. They are related to terms like "non monotonic reasoning", "truth maintenance systems", or "circumscription" (see [Doy79], [MCa86], [MDD80]). In addition to allowing more intuitive models, those concepts support the convenient maintenance of a knowledge base (although this way of maintenance does not have to result in satisfactory systems).

In order to get an idea of useful principles to guide overriding, we will consider two subjects of redefinition: classes and multiplicities. The examples focus on associations. Later, the decision that are illustrated through these examples will be generalized for attributes and services as well.

Classes

According to our experience, reconstructing common concepts often results in the need for redefining classes within inherited associations. Fig. 6 gives an intuitive example: A bicycle is composed of wheels. The association holds for all subclasses, except for RacingBike and RacingWheel: A racing bike must have racing wheels while a racing wheel can be mounted on a racing bike only. Fig. 6 reveals an additional problem: The graphical notation is misleading. It is not evident, whether the association between Bicycle and RacingWheel denotes an additional association or whether it redefines the association between Bicycle and Wheel.

.

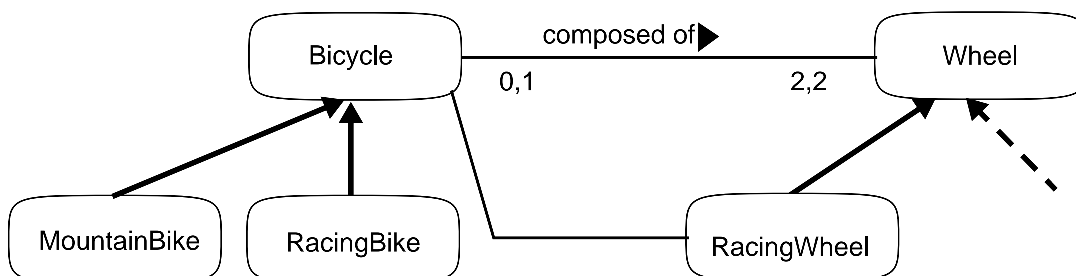


Fig. 6: Specializing a class within an association

While the notation is not our primary concern at this stage of our investigation, it is obvious that there is need to enhance the graphical notation. Fig. 7 shows a possible enhancement. Notice that there would be still need for yet another notation to render a partial redefinition of an

inherited association. For instance: While a racing bike requires racing bike wheels, it may be possible to mount racing bike wheels on any other bicycle (this case will be covered by the final notation, see 5.8.2).

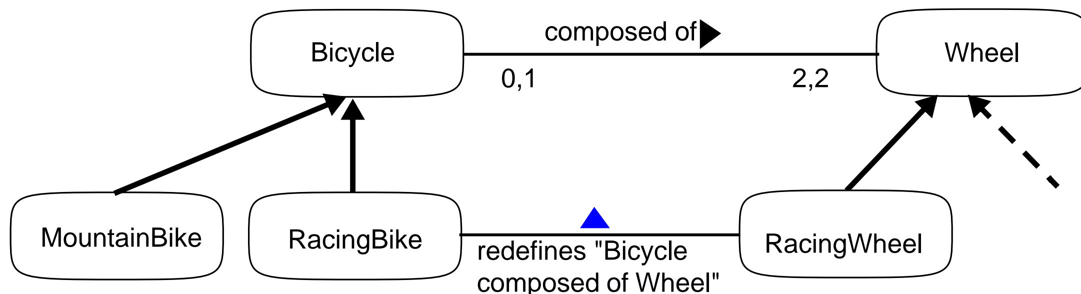


Fig. 7: Specializing a class within an association with corresponding notation

The redefinition shown in fig. 6 and 7 is somethimes called "covariant" ([Cast95], [Mey97], pp. 621). It denotes a rule of specialisation where features (or associated objects) of specialised classes may be redefined as specialisations (a RacingWheel is a specialisation of Wheel). While the covariance rule seems to correspond to the common sense use of specialisation, it implies a serious problem. From a logical point of view, the covariance rule leads to contradictions in a model's statements: "A racing bike is a bicycle" AND "A bicycle uses wheels" implies that a racing bike uses wheels which includes wheels that are not racing bike wheels. Hence, redefining the statement "A racing bike is composed of wheels" by "A racing bike is composed of racing bike wheels" results in a contradiction - with a devastating effect on a model's integrity. From a software engineering point of view (which is stressed, for instance, by [Mey97]), applying the covariance rule can result in type errors at run time. For example: An object of the class RacingBike receives the message addWheel with an instance of the class Wheel as an argument. Being kind of Bicycle, the object should act as any instance of Bicycle. Hence, the service addWheel should except a parameter of the class Wheel.

There is an alternative rule to redefine features of a class that avoids the problems of covariant redefinitions: The contravariance rule ([Mey97], pp. 626) prescribes that classes of inherited features may be overriden by one of their superclasses (generalisation instead of specialisation). Therefore, contravariant redefinitions seem to be preferable - at least from a software engineering point of view. However, despite its formal advantages, the contravariance rule seems of little use - you can hardly find an example that could be represented by using contravariant redefinition. Therefore, MEMO-OML allows for covariant redefinitions of inherited features only. Contravariant redefinitions would be easy to handle, but they do not seem to make any sense. Despite his preference for formal rigour, Meyer recommends the covariance rule as well - sacrificing "mathematical elegance" for "realistic and useful" designs:

> "An argument often encountered in the programming literature is that one should strive for techniques that have simple mathematical models. Mathematical elegance, however, is only one of several design criteria; we should not forget to make our designs realistic and useful too. In computing science as in other disciplines, it is after all much easier to devise dramatically simple theories if we neglect to make them agree with reality." ([Mey97], p. 626)

On a conceptual level, contradictions that can be produced by applying the covariance rule are hardly acceptable. In order to allow for exceptions without producing logical contradictions, classes of associated objects are regarded as true (in a rigorous sense) only if they do not contradict any redefined specialisation. Otherwise, they are valid for a restricted set of classes only. In other words: They are of limited generalisation. MEMO-OML also allows to define a class within an association as non-specialisable. In this case, it must not be redefined on a subclass level. Such a convention is similar to formal systems that have been introduced for "non monotonic reasoning". There, a proposition can be assigned the meta predicate "valid, if consistent" [MDD80]).

While MEMO-OML allows for specializing classes within an association, we do not encourage the use of this concept. This is at least for two reasons. First, it can be misused to "repair" generalisation hierarchies which were not carefully designed - thereby it would contribute to inappropriate abstractions. Second, it has to be taken into account that implementing such a concept may result in additional problems - especially with programming languages that do not feature dynamic typing. The notation used within the examples will be refined. However, it is hardly possible to provide a notation that allows to completely cover the entire semantic variety to occur within the specialisation of associations. Therefore the use of additional constraints may be required in some cases.

Multiplicity

Redefining multiplicities within an inherited association is a delicate task. This is partially due to the fact that inheritance as it is featured by most object-oriented programming languages (and as it is defined within MEMO-OML as well) has a meaning different from the common sense semantics of specialisation. At first sight, the multiplicity in example in fig. 8 seems to perfectly reflect the common understanding of roles people can hold: A person may act in none to many roles. A role can be regarded as an abstract generalisation over concrete roles such as student, professor, etc. By default, the multiplicity of an inherited association remains unchanged. This implies that the multiplicity assigned to Role (0,*) also applies for the specialised classes, such as Student or Professor. Hence, a person could be both a student and a professor many times.

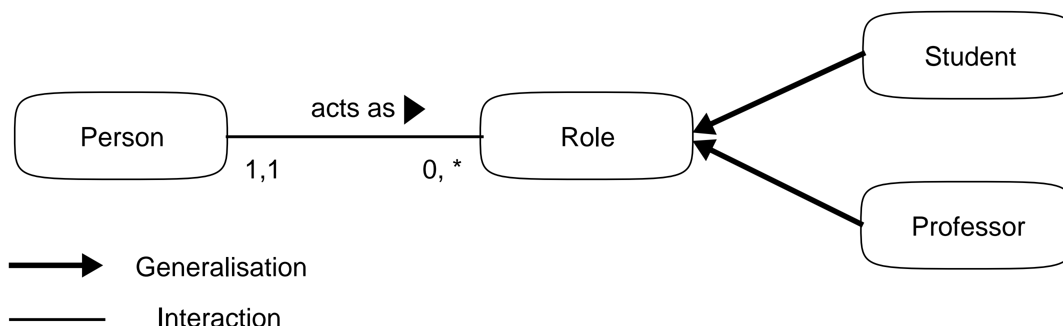


Fig. 8: By default, the multiplicity assigned to classes within an association is inherited to be used with subclasses, too. Hence, in this example a person could act many times both as a professor and as a student.

Consider now the following case: The model should express that a person can hold any role of

a particular type no more than once (in other words: it should not be associated with more than one instance of a class specialised from Role). Assigning a corresponding multiplicity - 0,1 - to Role seems to satisfy this constraint: All specialised classes would inherited it, and this is exactly what is required. However, at the same time this multiplicity would imply that a person cannot hold more than one role (of any type) in total (fig. 9). In order to apply such an assignment in a consistent way, it is necessary to introduce an implict constraint. The sum of the maximum cardinalities inherited by the subclasses of Role would be larger than 1 - contradicting the maximum cardinality assigned to Role. Therefore, we use the implicit constraint that maximum cardinalities of specialised classes within an association have to be interpreted in a context sensitive way, hence with respect to the actual cardinalities of other relevant subclasses. Notice that the implementation of this implicit constraint can result in a remarkable effort.

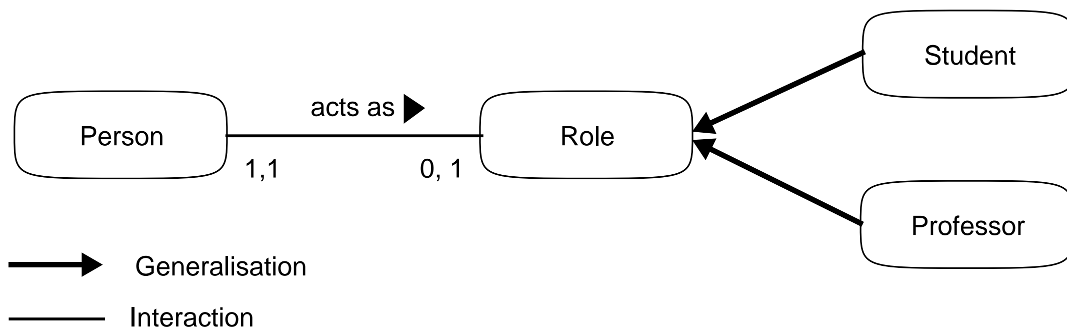


Fig. 9: The multiplicity assigned to Role - 0,1 - also applies for Student and Professor. Since no more than one role can be assigned at one point in time, a person cannot act as a student or a professor simultaneously.

Replacing the multiplicity in fig. 9 with 1,1 would result in a model that is not consistent anymore (see fig. 10): Inheriting this multiplicity to Student and Person would result in a minimum cardinality of 2 for Role. Therefore it is not allowed to use the multiplicity 1,1 for a class within an association that is subject of further specialisation - at least not without applying additional measures. To be more general, these considerations imply the following constraint: The sum of the minimum cardinalities inherited to the subclasses of a class that participates in an association must not exceed the maximum cardinality of the class itself.

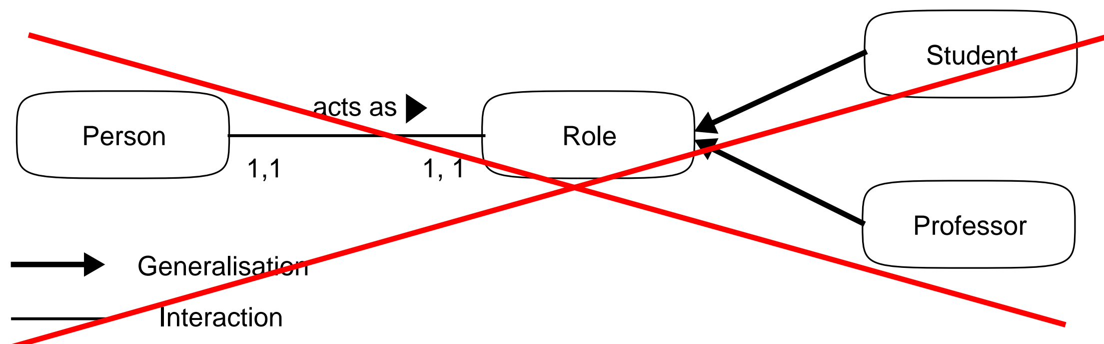


Fig. 10: Using the multiplicity 1,1 for a class that has subclasses in the way shown in this example is a contradiction in itself. For this reason it is not permitted - regardless of the domain.

Against the background of the interpretations introduced so far, it becomes evident that there is no straightforward way to express the case that a person may hold many roles at a time, but only one of a particular type. For this purpose we would need additional concepts. One option would be to specify a constraint that expresses the semantics required for a particular case. In principle, MEMO-OML allows for such an approach. However, since cases like the last one occur frequently, we decided to offer specialised concepts - together with a corresponding notation - to express various kind of specialising associations.

In order to avoid ambiguity (and contradictions which would compromise a model's integrity), the affected multiplicities have to be specified as *redefined*. The redefinition of multiplicities is restricted to certain modifications. The multiplicity of a subclass may only be specified by a range which is included in the range of the superclass. This rule is due to the common sense interpretation of specialisation: An instance of a subclass is an instance of a superclass at the same time. Although this is not the case for the concept of specialisation used within MEMO-OML, the rule is enforced in order to foster an intuitive understanding of models. In addition to that, it is also required that the constraint concerning the sum of the minimum cardinalities applies in this case, too: The sum of both, inherited and redefined cardinalities must not exceed the maximum cardinality assigned to a superclass - this does not only include the direct subclasses but all subclasses (a more precise specification of this constraint will be provided later with the metamodel).

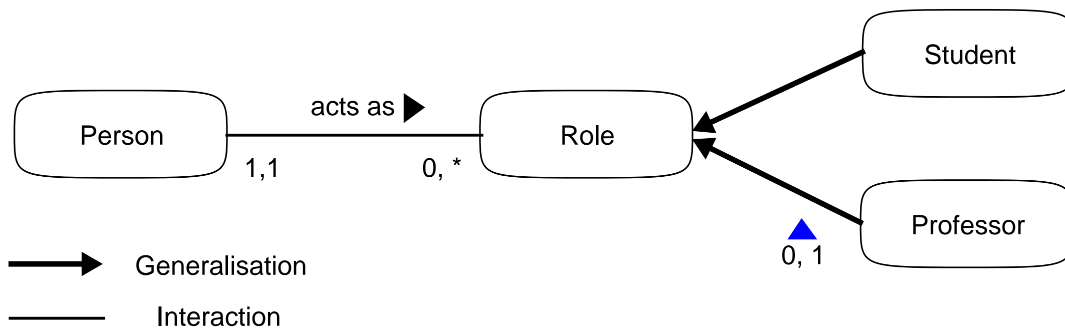


Fig. 11: Specialisation of multiplicities within an association are permitted. The affected multiplicities have to be marked as as specialised (this is a preliminary notation).

Attributes and Services

Associated objects are accessed via services. Therefore, the rules defined for redefining inherited associations have to apply for services as well. While there is a clear difference between attributes and associated objects on a conceptual level, this difference depends on the context and will usually vanish on the implementation level where both, associated objects and attributes, are represented as instance variables. Because of this strong similarity, we decided to use the same rules for redefining the class and multiplicity of inherited attributes as we have introduced for associated objects. In any case, it is required to characterize an overridden attribute or service as redefined. Fig. 12 illustrates the rules to be applied for redefining attributes and services.

Attribute

Feature	*Specified by*	*Redefined by*
Class	C	C’
Multiplicity	min, max	min’, max’

Service

	Feature	*Specified by*	*Redefined by*
ReturnedObject	Class	C	C’
	Multiplicity	min, max	min’, max’
Parameter	Class	C	C’
	Multiplicity	min, max	min’, max’

Rule:

C’ is subclass of C ("covariance")
min’ >= min
max’ <= max
min’ <= max’

Fig. 12: Redefinition of Attributes and Services

Subtyping

Sometimes it may be necessary to ensure that an instance of a class can be replaced by any instance of one of its subclasses. In this case, it is common to speak of *subtyping* as a special form of inheritance. Subtyping implies that the covariant redefinition of features is not applicable. Since the contravariance rule does not make any sense, we speak of subtyping only if none of the inherited features (or associations) has been redefined. Hence, subtyping is an implicit concept within MEMO-OML. The only way to express that a class should by a subtype of a superclass would be to explicitly forbid redefinition for all of the superclasses’ features (and associations). This can be done using a corresponding attribute for any feature. This implies that a class cannot have a subclass and a subtype at the same time.

Delegation

In many application domains there are certain aspects that cannot be modelled in an adequate way by using generalisation or common associations (like interaction or aggregation). In those cases *delegation* often proves to fill this conceptual gap.

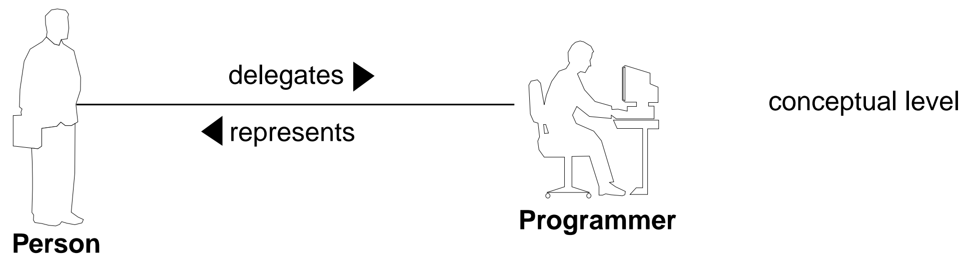


Fig. 13: Meaning of Delegation on the Conceptual Level

From an implementation point of view delegation is essentially based on transparent message dispatch. We define delegation as a special association with the following general characteristics:

1. Delegation is a *binary* association with one object (the "role" or "role object") that provides transparent access to the state and behaviour of *another* (not the same) object (the "role filler" or "role filler object").
2. The role object dispatches every message it does not understand to its role filler object. Thereby, it does not only dynamically "inherit" a role filler object's interface (as it would be with inheritance, too) but also represents the particular role filler's properties. In other words: It allows for transparent access to the role filler's services *and* state. In case a role filler object includes a service that is already included in a role object's native interface (defined in its class or one of its superclasses), the role object will not dispatch the message to the role filler object. Instead the corresponding method of the role object is executed.

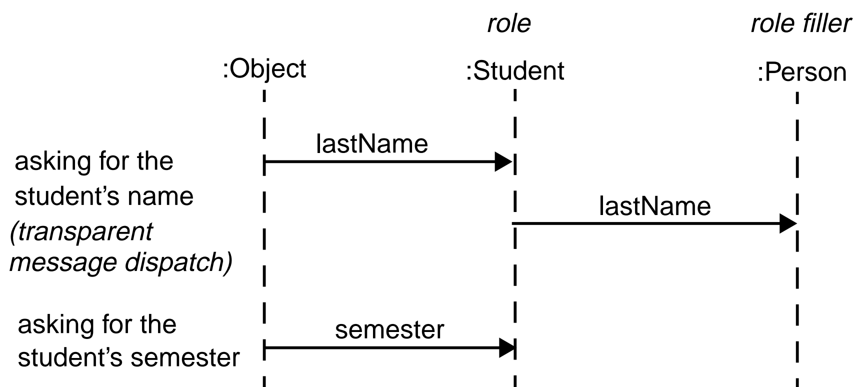


Fig. 14 Sequence Diagram to illustrate the Dynamics of Delegation

3. Inheritance and delegation: Both, the responsibilities of a role filler class and a role class are by default inherited to their respective subclasses.
4. A role filler may in general have none or many roles. For a particular delegation, the multiplicity of roles can be specified within this range. A role filler may have more than one

roles of the same class. For instance: An object of the class Person may be associated with more than one instance of the class Programmer at the same time - a programmer with Smalltalk experience and another one with C++ experience (that does not mean, however, that we would recommend to always use two instances for modeling this situation).

Different from the less restrictive use of the concept in some delegation based programming languages, we propose a number of constraints:

#1 *Only classes that are kind of a special role class or a special role filler class can be used to serve as roles or role fillers within a delegation association.* This is for two reasons: Not any object is conceptually suited to serve as a role or a role filler respectively. Moreover, the special semantics of both classes will often require certain extensions on the implementation level.

#2 *The number of role filler classes to be used for a particular role class is restricted to one.* While there are real world situations where it seems to be appropriate to have a role class associated with more than one role filler classes (see example 2 below), such a "multiple delegation" would substantially decrease the chances to check a model's integrity. The concept of delegation we have decided for does not allow for multiple delegation, since we regard integrity a more valuable asset than flexibility in this case. That does not necessarily exclude to have a role associated with instances of different role filler classes - provided they are all subclasses of one common superclass. It may be helpful to define an abstract superclass for this purpose, thereby providing a minimum common protocol for all possible role fillers (see example 2 below).

#3 *At a point in time, a role object must not be associated with more than one role filler object.* While associating a role object with more than one role filler object of the same class (#2) would not add confusion with respect to the interface, it would certainly jeopardize the whole idea of delegation: A role represents exactly one role filler and allows transparent access to that role filler's state. Notice that this does not exclude a role object to change its role filler object over time.

#4 Multi-level delegation is possible. However, *cyclic associations are not permitted.* Since the number of a role class' corresponding role filler classes is restricted to one, it seems appropriate to allow a role object to also act as a role filler object (which one might call multi-level delegation): It may increase a model's complexity but it is no serious threat to its integrity. For this reason, multi-level delegation is not excluded by our definition of delegation. By no means may a role object act as a role filler of itself: In most cases, one would regard an object that is a role of itself as a bizarre abstraction on a conceptual level. On an implementation level, a cyclic association of this kind would impose the threat of non-terminating message dispatches.

It is up to the experienced analyst who should always be in charge of modelling to guide the domain experts with identifying possible delegation associations. For instance: Whenever a specialisation could be replaced by an association labeled "represented by" or "delegates", it is usually a good idea to choose delegation instead of generalisation/specialisation (for a detailed description of such guidelines see [FrHa97]).

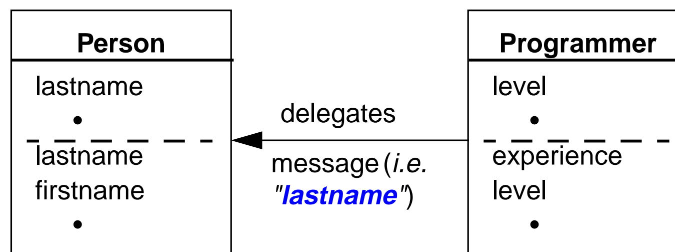


Fig. 15: Delegation from an implementation point of view.

4.3 Additional Concepts (Object Design)

Beside the concepts that serve to refine classes, there is one additional general concept: A class may be an instance of exactly one *metaclass*.To allow for a more detailed specification of a class within object design, MEMO-OML offers additional features to specify attributes and services as well as two additional *semantic class features*: *guards and triggers*. Associations, attributes, and services can be defined as "deferred" which means that they have to be specified within a subclass. An association is regarded to be deferred if at least one of the two association links it belongs to is deferred. Only abstract classes may have deferred features. Referring to Meyer, we call an abstract class that has a deferred feature a *deferred class* ([Mey97], p. 486). A feature that is not deferred is called "*effective*" ([Mey97], p. 486). Each semantic class feature may have any number of instances. Each instance needs to have a unique name within the scope of the class it is assigned to.

4.3.1 Attributes/Associations

For its use during design, an attribute can be specified further by a number of additional features. In case the maximum cardinality is larger than one, an additional predicate can be assigned (like "ordered", "sorted"). Notice that there are only two semantic differences between an attribute and an associated object. First, an object that serves as an attribute must not be of the same class as the class of the encapsulating object or any of its subclasses. This constraint is thought to prevent problems that could occur with non terminating recursive initialisations. Second, the association between an object and its attribute is not symmetric. In other words: An attribute must not have its object as an attribute. Attributes and associated objects are differentiated to express different intentions of the language user: While an object (or class respectively) represents an entity which has an identity of its own within an object model, an attribute represents an entity that has no such identity outside the scope of the object it is encapsulated in. The following features can be assigned to both attributes and associated objects. *AccessType* serves to specify the scopes in which particular modes of access may occur. AccessType is specified by a set of access modes and corresponding privileges. Access mode is either #read or #write, or - only in the case of a maximum cardinality larger than one - #remove or #add. Each access mode has a *Privilege* assigned which is either #private, #public or #protected. #private indicates that the attribute is only to be accessed from within the encapsulating object. If only authorized objects are allowed to access the attribute, this is indicated by #protected. Finally, #public expresses that the attribute can be accessed by any object. Notice, that this specification deviates from the use of access privileges in C++. In the case of an external access, an appropriate service is required to allow for the access.

In addition to its semantic features, an attribute has a number of management features. D*efault-Value* is specified by an instance of the attribute's class and serves to indicate the state an attribute should have by default. *History* is specified by a boolean value. It indicates whether or not the changes of an attribute's state are to be recorded (for instance: when the name of a customer changes, you may want to record the previous name). Any of an attribute's or an association link's management features (*History*, *DefaultValue, AccessType*) can be redefined within a subclass. This is also the case for an exception assigned to a service.

4.3.2 Services

Access rights define the scope in which the service may be used (#public, #protected, #private). Their meaning corresponds to that defined for attributes (see above). In case the implementation language does not support a particular access right, it may be reasonable to use a slightly deviant interpretation.

We differentiate five categories of services: *regular* services, *default access* services, *propagated* services, *deferred* services and *redefined* services. Default access services are services which provide default access (read, write, insert, delete) to an object's attributes (*AttriAccessService*) or its associated objects (*AssocAccessService*). Regular services can be specified by the attributes (of the same class) or services (of any class) they use. Propagated services mainly serve to provide an abstraction: They are propagated from the interface of an attribute of the particular class. For instance: A class may have an attribute "dateOfBirth" which provides a service "yearsOfAge". A propagated service allows to make that service available directly within the interface of the corresponding class (like "Person") without loosing track of the reference. A deferred service can be assigned to an abstract class only. It must not be implemented within the class it is assigned to. Any service may be assigned a *precondition* and a *postcondition* which are expressions to be specified in a language still to be defined. Preconditions and postconditions may refer to any object or service, as well as to attributes of the same object. *Control* allows to specify how a service fulfills its task. In the current version, there are not special language concepts to support this specification. It could be done with a formal specification language, pseudo-code or a semi-formal graphical language (like state charts). It can only be assigned to regular services, since the purpose of other service types is predefined already. The responsibility of a class is implicitly defined by the pre- and postconditions of its services. A redefined service redefines a service inherited from a superclass. The constraints that apply to the specialisation of services will be explained later (see 5.1).

In general, the signature of a *redefined service*, the parameter(s) as well as the class of the returned object may be modified: The parameter name(s) may be changed, their class(es) as well as the class of the returned object may be redefined according to the covariance rule. Signatures within redefined default access services must not be modified deliberately: Both parameters and returned object are determined by the attribute/associated object that is accessed. In case the multiplicity of an associated object is redefined as a multiplicity with a maximum cardinality smaller than two, access services that depend on the existence of more than one associated objects do not make sense any more. However, since it is not possible to remove an inherited service, the only way to handle this case is to make the corresponding services (like adding an associated object) ineffective in an appropriate way.

Preconditions, postconditions within inherited services are a delicate subject. As long as there is no formal language to specify preconditions and postconditions, the following informal rule applies for corresponding redefinitions: Any redefinition of inherited preconditions and post-

conditions is allowed as long as it does not compromise the rules defined for redefining parameters and returned objects.

At first sight, it may appear that default access services are of no relevance unless you want to generate code. However, in order to add more specific pre- or postconditions or for the specification of guards or triggers, it may be necessary to refer to those services. Furthermore, it has to be taken into account that additional models, like workflow models, may use particular default access services.

In addition to its semantic features, each service can be characterized by a management feature*: Exception* serves to identify exceptions which may occur during execution.

4.3.3 Guards and Triggers

A trigger serves to express a rule of action in the responsibility of the objects of its class. The rule consists of an event and a corresponding action which is performed by a particular service of that class. An event is specified by a boolean expression. The event occurs when the expression evaluates to true. It can be caused by a state change (in any set of objects) or by the progress of time (which may be regarded as a state change as well - for instance: whenever an employee turns 50, his salary has to be increased by 3%). It should not be used to express a rule that could also be specified by a precondition or a postcondition. A trigger may refer to any object or service, as well as to attributes of the same object.

A guard is an invariant (see [Mey97], pp. 364) which goes beyond the scope of a service or an attribute. It serves to prevent inconsistent states. For example: "Make sure that the retail price of a product cannot be lower than its wholesale price." As with preconditions, postconditions and triggers, guards are specified in a language that has not been defined/selected yet. They may refer to any object or service, as well as to attributes of the same object. A formal specification of these concepts is required in order to avoid any contradictions between them (and between them and other concepts within an object model). A guard is more specific than a constraint in the sense that it is a feature of one class only, i.e. as soon as this class is removed from the object model, the guard does not make sense any longer. Both guards and triggers of a class may be assigned a name which should be unique within the class.

Inherited guards and triggers may be redefined (see fig. 25). This is a preliminary regulation which may require additional constraints in the future. On the implementation level, guard and triggers are usually reconstructed as services. In case a redefined guard implies that an effected service gets modified or becomes obsolete, those changes have to be applied as soon as the guard is in place - otherwise the integrity of the corresponding class would be harmed.

4.3.4 Constraints

A constraint can be assigned to any modelling element. It should, however, be used only if other, more specific concepts (like guards, pre- or postconditions) are not applicable or less appropriate. Examples: "Within the object model there must be a class that provides a service xy."; "The generalisation hierarchy must not be deeper than 8 levels." A constraint must not contradict any other concept within an object model.

4.3.5 Subtyping (Refinements), Interfaces, Metaclass

Sometimes the term *interface* is used to allow for an additional abstraction. If you want to make sure that certain classes within different object models offer a specific set of services,

you may want to specify these sets without regard to a particular class. Think, for instance, of basic services to be provided by a text editor, like copy, cut and paste of text. Another example would be the elements of a standardized software architecture - like the interfaces that have to be provided by parts implemented within an OpenDoc environment ([Ope94]). Any interface that has to be satisfied within the scope of an object model could be modelled as an abstract class. Since MEMO-OML allows for multiple inheritance, the class that should provide a particular interface would be defined as subclasses of the corresponding abstract class. In case you have to map an existing interface to one requested by a specification, you would have to define a class that provides the requested interface together with the dispatching to the existing interface. In other words: MEMO-OML does not explicitly offer the concept of an interface. It can, however, be represented by other existing concepts.

Our interpretation of "subtyping" and "interface" is different from the concepts "type" and "interface" proposed by the UML. Notice, however, that the use of these concepts is not consistent within the official documents about the UML - only part of which is due to changes that occured between different versions of the language. Version 1.0 includes the following statement:

> "Class is a subtype of Type, and therefore instances of Class have the same property as instances of Type. The fundamental difference being that Type instances specify interfaces, whereas Class instances specify the realization of these interfaces." ([Rat97a], p. 57)

Later, in version 1.1, both concepts are defined in a different way:

> "An interface is the use of a type to describe the externally-visible behavior of a class, component, or other entity (including summarization units such as packages)" but also: "An interface is a type and may also be shown using the full rectangle symbol with compartments" ([Rat97g, p. 25])

> "A Type characterizes a changeable role that an object may adopt and later abandon. ... . An object may have multiple Types (which may change dynamically) but only one ImplementationClass (which is fixed)" [Rat97f], p. 35)

We do not think it is appropriate to introduce types within object models because they are not required (for a similar point of view see [Mey97], p. 24). Instead they may cause confusion. In any case we do not agree with the authors of UML to define a type without regard to its semantics. We rather speak of classes - some of which may be specified by subtyping them from other classes. As already explained abobe, MEMO-OML allows to specify a subtyping relation in an indirect way - by defining all the inherited attributes, services and associations of a class as not specialisable. It is the purpose of this rule to ensure that an instance of a subtyped class can replace any instance of its superclass (replacement rule). Different from the general concept of inheritance, this implies that inherited pre- and postconditions must not be changed either. However, in order to allow some sort of flexibilty without violating the replacement rule, it is possible to redefine the algorithm (control) of an inherited service (and, at a later stage, to introduce a corresponding implementation).

While it is arguable whether it makes sense to regard classes as objects within conceptual modelling, this idea can be useful to specify information which is required for system design. For this reason MEMO-OML offers metaclass as an optional concept. A metaclass is a class with exactly one instance which has to be a class. A class in turn must not have more than one metaclass.

4.3.6 Organisation of Object Models

Object models can become rather complex. This recommends organisation concepts which allow to reduce complexity. At present time, MEMO-OML offers only one concept which allows to group classes of an object model. Each class can be assigned zero or one *category*. A service can be assigned to one *protocol* which needs to have a unique name within one class. A protocol may contain zero to many services. An attribute group serves the same purpose for attributes. It must have a unique name within a class, and it may contain zero to many attributes. Although they do not have any semantic impact, inherited protocols must not be changed in order to avoid confusion. Constraints that effect a certain class may be redefined by constraints introduced for subclasses. This preliminary rule is due to the fact that MEMO-OML does not include a formal constraint specification language yet.

In addition to those grouping concepts, MEMO-OML includes a organisation concept that serves a similar purpose. A *cluster* is a collection of classes that provides an identifiable function within a system. Different from categories a cluster may contain other clusters. Furthermore, a class may be part of more than one cluster. This notion of a cluster is inspired by a modelling method called "Business Object Notation" [WaNe95]). Associations between clusters can be derived from the associations between their classes. Clusters allow to split a complex system in a set of interacting units which may be decomposed further. The graphical representation of the clusters that constitute a system can be regarded as a system architecture. Notice that this does not imply any restrictions on the level of abstraction you may want to choose for a particular architecture.

In recent years, object-oriented frameworks have gained remarkable attention for their potential to facilitate large scale reuse. According to Cotter and Potel "A framework embodies a generic design, comprised of a set of cooperating classes, which can be adapted to a variety of specific problems within a given domain" ([CoPo95], p. xx). In the most simple but least flexible case, a framework does not allow for individual modifications. Instead its functionality can be accessed through the interfaces of a set of classes which are part of the framework. Different from this "use as is" strategy, within the "complete" strategy a framework includes parts - like abstract classes - which require further specifications, typically through specialisation. Finally, there are frameworks which allow to redesign/override certain parts ("customize", [CoPo95]). Those parts of a framework which are visible for modification are sometimes called "hot spots" where the rest of the framework consists of "frozen spots" ([LaNa95]). On the level of abstraction appropriate for object modelling, we regard a framework as a collection of associated classes with a number of additional features:

- any class within a framework has to be implemented
- any class within a framework can be visible for reuse or not
- visible classes may be specialised or not
- visible classes may be modified or not
- visible classes may be replaced or not

Notice that the modification of a framework should not hurt any integrity constraint imposed by the framework itself. Since this aspect is out of scope, we do not regard it any further.

Similar to frameworks, design patterns are often regarded as a powerful concept to foster reuse. In its original sense, a design pattern provides a structured description of "good" design

for a class of problems ([GaHe95]). Hence, design patterns can be regarded as a contribution to a study of (object-oriented) modelling. From a slightly different point of view, design patterns can be regarded as a technique to describe a generic design, i.e. a design that is suitable for a number of particular problems. In this case, emphasis is on documentation. For this reason, design patterns have been recommended for an illustrative documentation of frameworks ([Joh92]). There is no unique definition of a design pattern's structure. Probably the most common structure is the one suggested by Gamma et al. ([GaHe95]).

- Pattern Name
- Also Known As
- Purpose
- Motivation
- Applicability
- Structure
- Participants
- Dynamics
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

The concept of design pattern provided by MEMO is based on this structure. Design patterns can be used to document the design of generic clusters or frameworks.

4.4 Implementation-Oriented Refinements (System Design)

In order to allow for a seamless transformation of an object model into an implementation, it is necessary to add information that is required on the code level. Furthermore, it can be a good idea to enhance the model with specifications that target implementation relevant issues like performance and persistence.

4.4.1 Stored References

In most implementation level languages, associations are implemented through variables that store references to associated objects. Within a binary association, at least one of the associated objects needs to store such a direct reference. If both associated objects store a reference, we speak of a bidirectional associaton. Otherwise we speak of a unidirectional association. In order to prepare a design model for implementation, MEMO-OML allows to assign an attribute to express whether or not the objects of the corresponding class should store a reference to the object(s) they are linked to through a particular association (see fig. 35).

4.4.2 Persistence

Many applications require persistent objects. However, usually not all objects described in an object model have to be made persistent. For this reason, it is possible for any class in a MEMO

object model to specify whether the lifetime of their objects should exceed the lifetime of an application instance. Considering the performance of corporate information systems, the time to access a piece of information can be of crucial importance.

4.4.3 User Interface Concepts

The user interface of a system to be developed is often regarded not to be subject of conceptual modelling, since a conceptual model should focus on features which are essential to the relevant domain. Nevertheless it can be helpful to include information that is relevant for the implementation of a user interface. This is for various reasons:

- *Productivity*: The design and implementation of the user interface consumes a considerable amount of the overall system development time. Enhancing an object model with information concerning the user interface may contribute to faster development cycles by allowing for reuse and code generation.
- *Quality*: The user interface can be a complex part of the entire system. An abstract description improves the chances for a well designed and consistent user interface.

Notice, however, that adding specific information to an object model is hardly efficient for a complete specification of a user interface: The interaction with the classes of a system may vary with the context. An object model, however, does not allow to express certain contexts of interaction with a system. Those contexts can be taken into account within other perspectives, like process or workflow models.

In order to prepare for a rudimentary model of a user interface, it can be helpful to assign a default user interface to every class in an object model. However, not every class will require an explicit assignment. Often, a user interface can be constructed from the user interfaces previously assigned to (or constructed for) classes of attributes, associated objects, parameters and returned objects. Within an object-oriented information system, the functionality of a user interface will usually be specified by an object model (or more likely: a framework of implemented classes). In other words: User interface concepts are rather an application of an object-oriented modelling language than part of it. Therefore, the preliminary classes we introduce for describing aspects of a user interface will be specified later as an object model using MEMO-OML (see 6). We do need, however, a few enhancements of MEMO-OML in order to allow for UI-specific enhancements of an object model (see fig. 36).

5. The Metamodel

Designing a metamodel for a language can easily become a frustrating endeavour. Often, there will be various options to represent a particular concept. Different from the design of a real world domain, there is usually no chance to evaluate those options against common perceptions of reality. Hence, the design of a language metamodel will always include decisions that are not only based on rational considerations but that also reflect subjective taste. It seems to be impossible to completely overcome this problem. There is only one way to cope with it: A metamodel should be explained in a way that it is possible to identify relevant desicisions and the assumptions/preferences they are based on. While this is the intention of the following section, we will not explain every concept of the metamodel since some concepts should be explained well enough by the metamodel itself (others have already been explained in the previous section).

Semantic concepts serve to describe those properties of an object which correspond to properties of real world entities or concepts. Within the metamodel *semantic* concepts are rendered in black. *Organisation* concepts are rendered in green, while *management* concepts are rendered in blue. Finally, *ressource* concepts are rendered in red colour.

5.1 Basic Concepts

All concepts within the Metamodel are specialised from Object. This makes it easier to add general modifications that apply to all concepts of the metamodel. ModelElement is an abstraction for all elements of an object model that are rendered as graphical symbols (or as part of those). Comment, Constraint and Expression are not specialised from ModelElement. This is for a simple reason: Constraints and expressions can be applied to any element of a model, except for constraints, expressions and comments. Notice that comments can be applied to any element, including other comments as well. Instances of Expression are intended to hold expressions of a (semi-) formal specification language that has yet to be defined. NamedObject provides a generalisation over all concepts that have a mandatory name which serves as an identifier within a particular context. Different from that, LabeledObject is a generalisation over those concepts that may have an optional label which serves only to improve the readability of a model. Tt present time, the metamodel contains one specialisation of LabeledObject, AssociationLink. Nevertheless, this abstraction was chosen in order to support further enhancements of the metamodel. Like associations, class features have to be specified in more detail. The generalisation hierarchy in fig. 20 only shows the highest abstractions of any class feature, BasicAttribute, BasicService, BasicGuard and BasicTrigger. Instance serves to model instances of Class. While one usually can do without instances in an object model, they are sometimes required, for instance to define default values. Classes are differentiated into concrete and abstract classes with GenericClass as a common generalisation.

Notice that those concepts within the metamodel that are not explicitly specialised from any other concept are assumed to be specialised from Object.

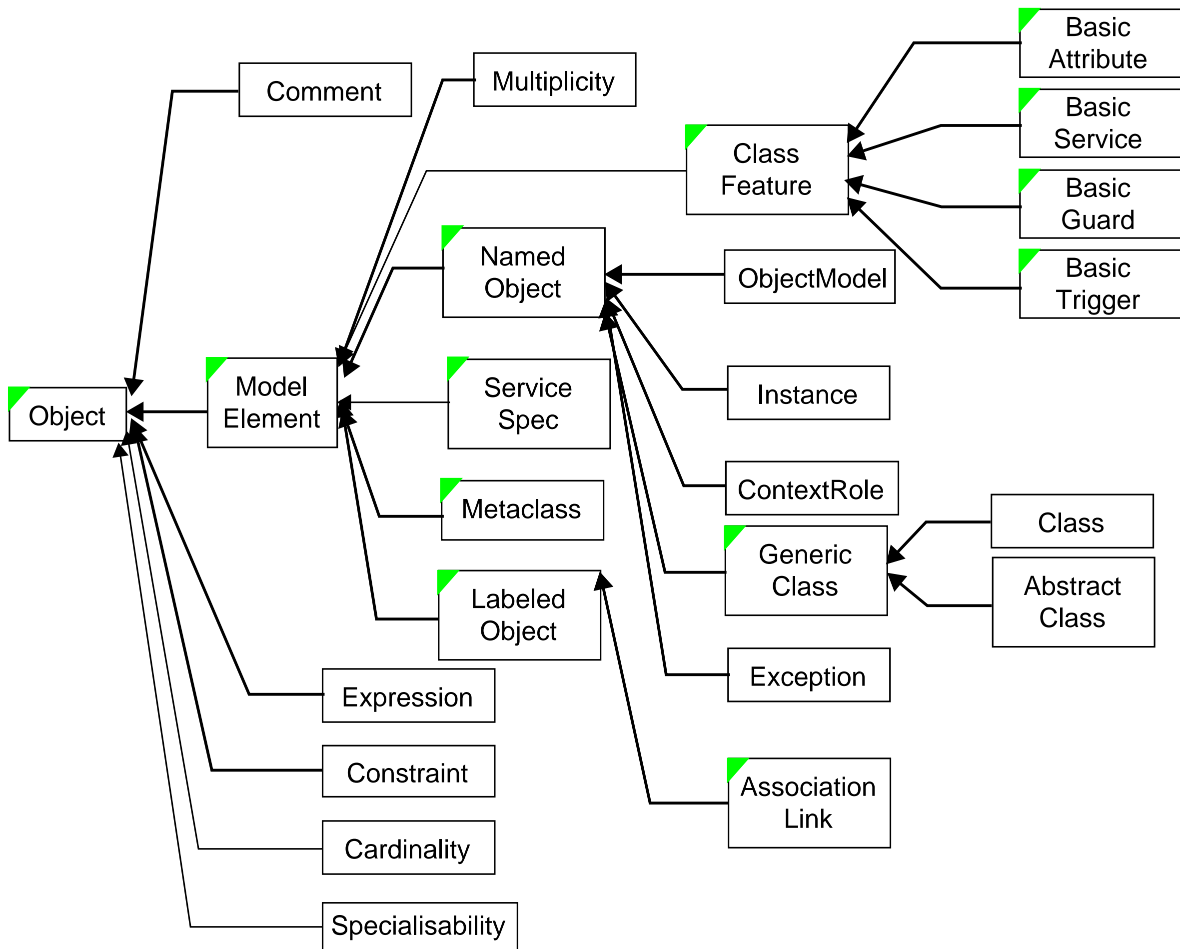
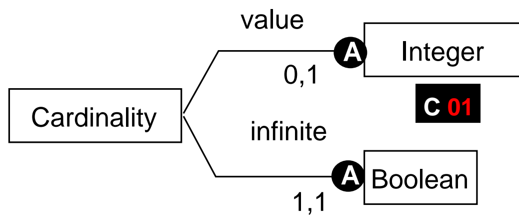


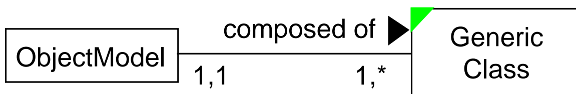
Fig. 16: MEMO-OML Metamodel: Generalization Hierarchy (to be extended)

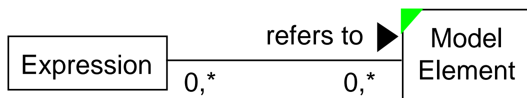
While an object model contains instances of any modelling concept (except for ObjectModel itself), it is sufficient to model it as composed of classes only. While comments and constraints can be related directly to an object model, any other modelling element is either part of a class or is associated with it. OrderSpec serves to express an order of objects. A particular order can be defined by picking one of four predefined symbols: #timeUp, #timeDown, #sortedUp, #sortedDown. This is a preliminary solution only. In order to ensure the integrity of an OrderSpec, it would be necessary to check whether a selected order can be applied to the corresponding objects. An instance of OrderSpec is assigned to Multiplicity and may be redefined within inherited associations or attributes. This may result in the necessity to redefine corresponding access services.



C 01 The attributes infinite and value must not be instantiated simultaneously.

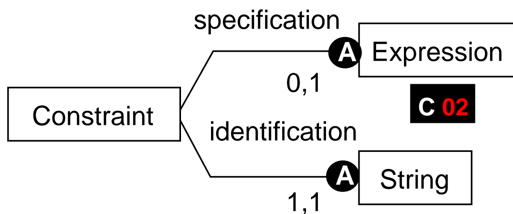
value >= 0





C 02 The specification of a constraint must must not contradict (may, however, override) existing concepts. It should be used only if no more specific concept are available.

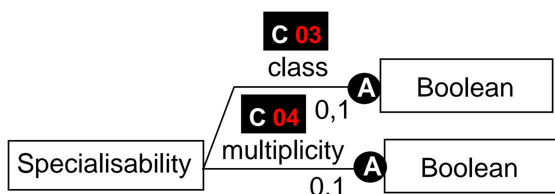
The identification has to be unique within an object model.



The specification of a constraint must be represented in a formal language still to be defined. At this point, we regard a valid expression in this language to be an instance of Expression.

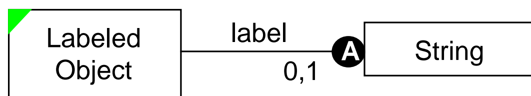
While it is not required, it is recommended to use positive numbers as identifiers.

The default value of both attributes, class and multiplicity, is "true" indicating that class and multiplicity of the corresponding object may be redefined.



C 03 The value of the class attribute has to be the same as the value of the class attribute of the associated AssociationLink.

C 04 Multiplicity may only be true if maxCard - minCard > 0.



The label of an instance of LabeledObject does not have to be unique within any scope.

Fig. 17: Basic Concepts (1)

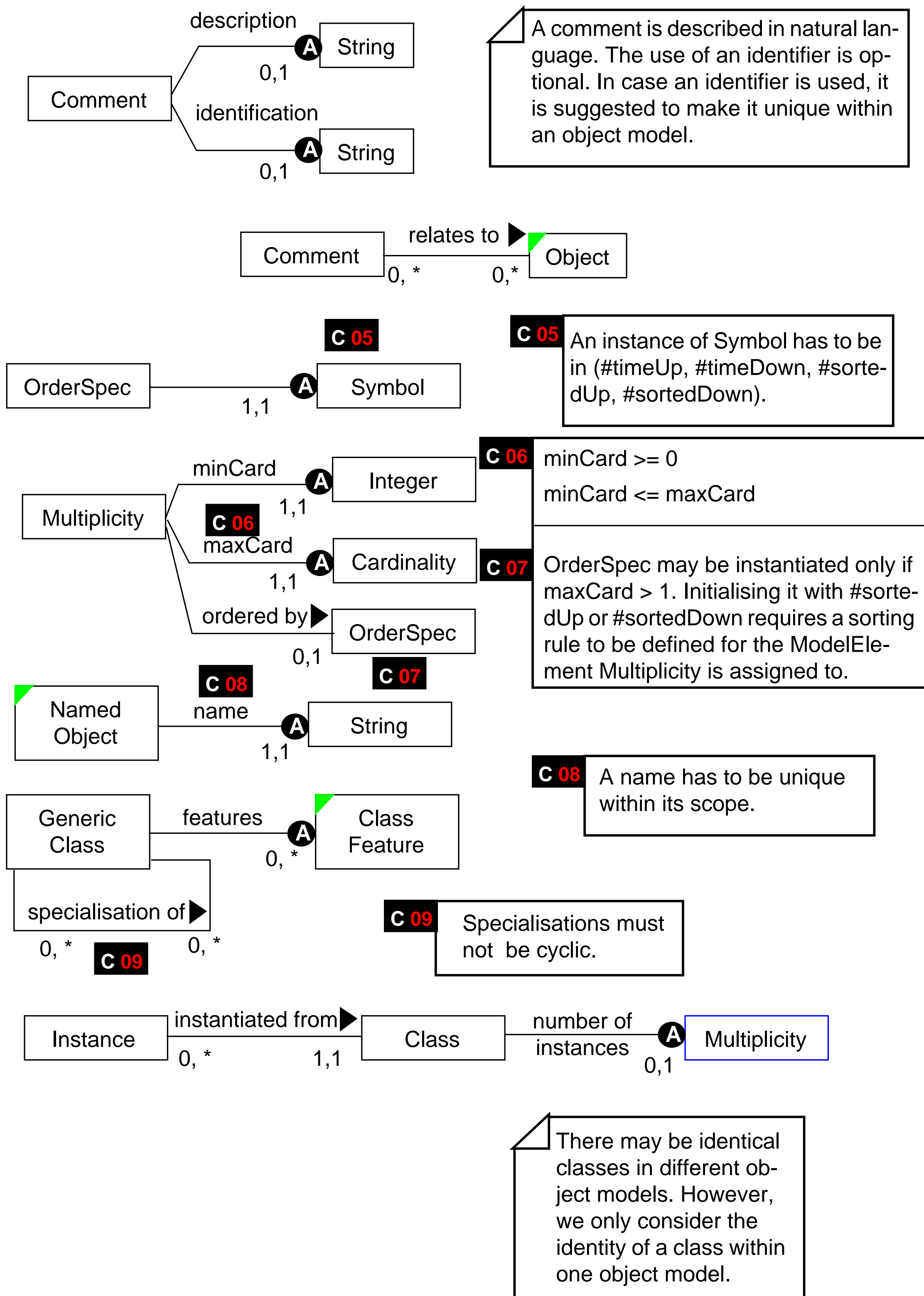


Fig. 18: Basic Concepts (2)

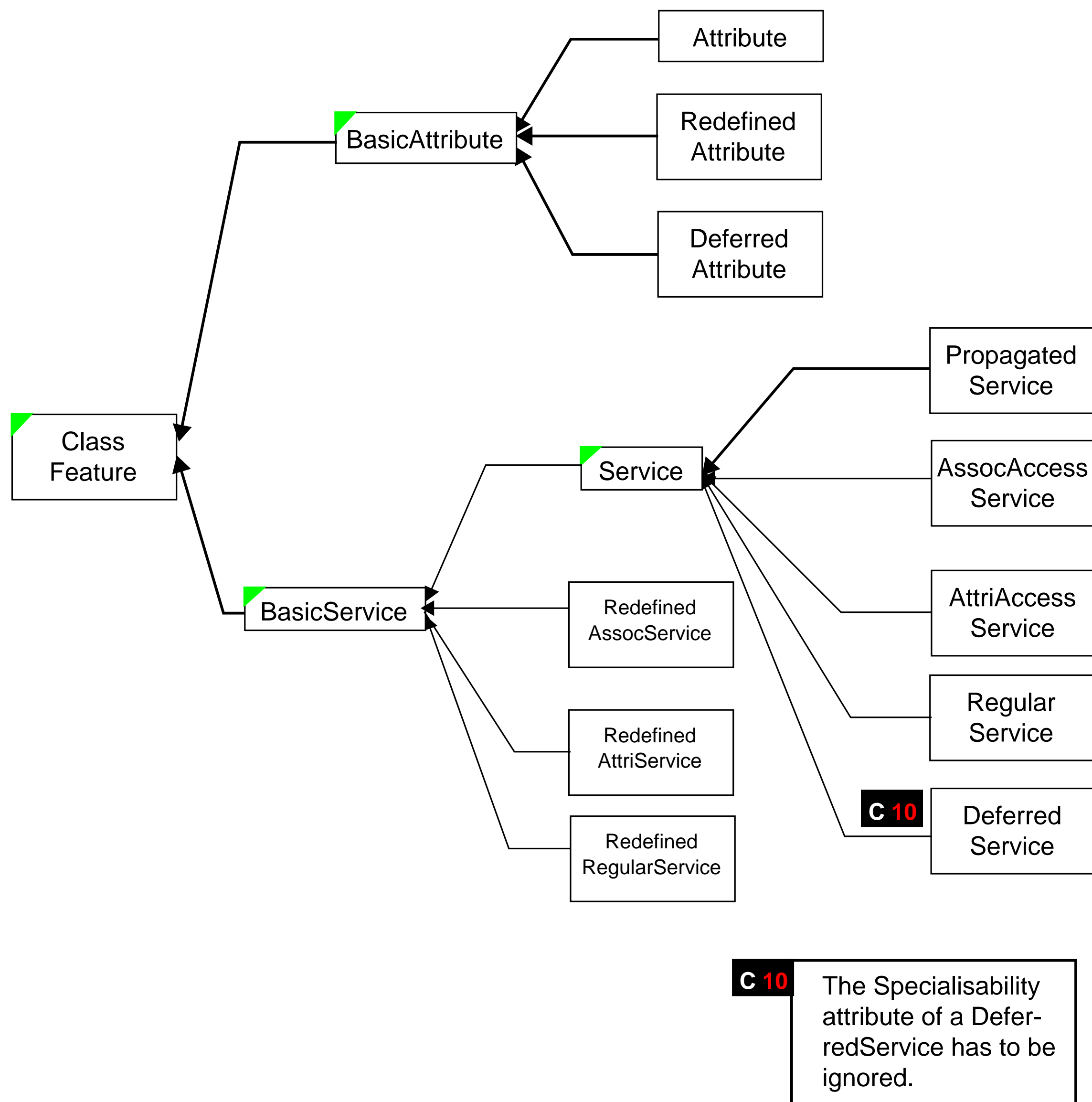


Fig. 19: Services and Attributes - Generalisation Hierarchy

The generalisation hierarchy rendered in fig. 19 shows the the various kinds of attributes and services described in 4.3.2. Among other things, their differentiation is motivated by the aim to express the semantics of inheritance on a syntactic level. AccessType allows to specify access rights for attributes and associated objects, differentiated into four modes of access: read, write, add and remove. Privilege, which can also be assigned to services, serves to specify one ot three access scopes: *public*, *protected* and *private* (see 4.3.1). The concepts represented in fig. 21 provide a specification of attributes and their possible redefinitions. Specialisability was not assigned to the general concept BasicAttribute since it does not make sense for DeferredAttributes.

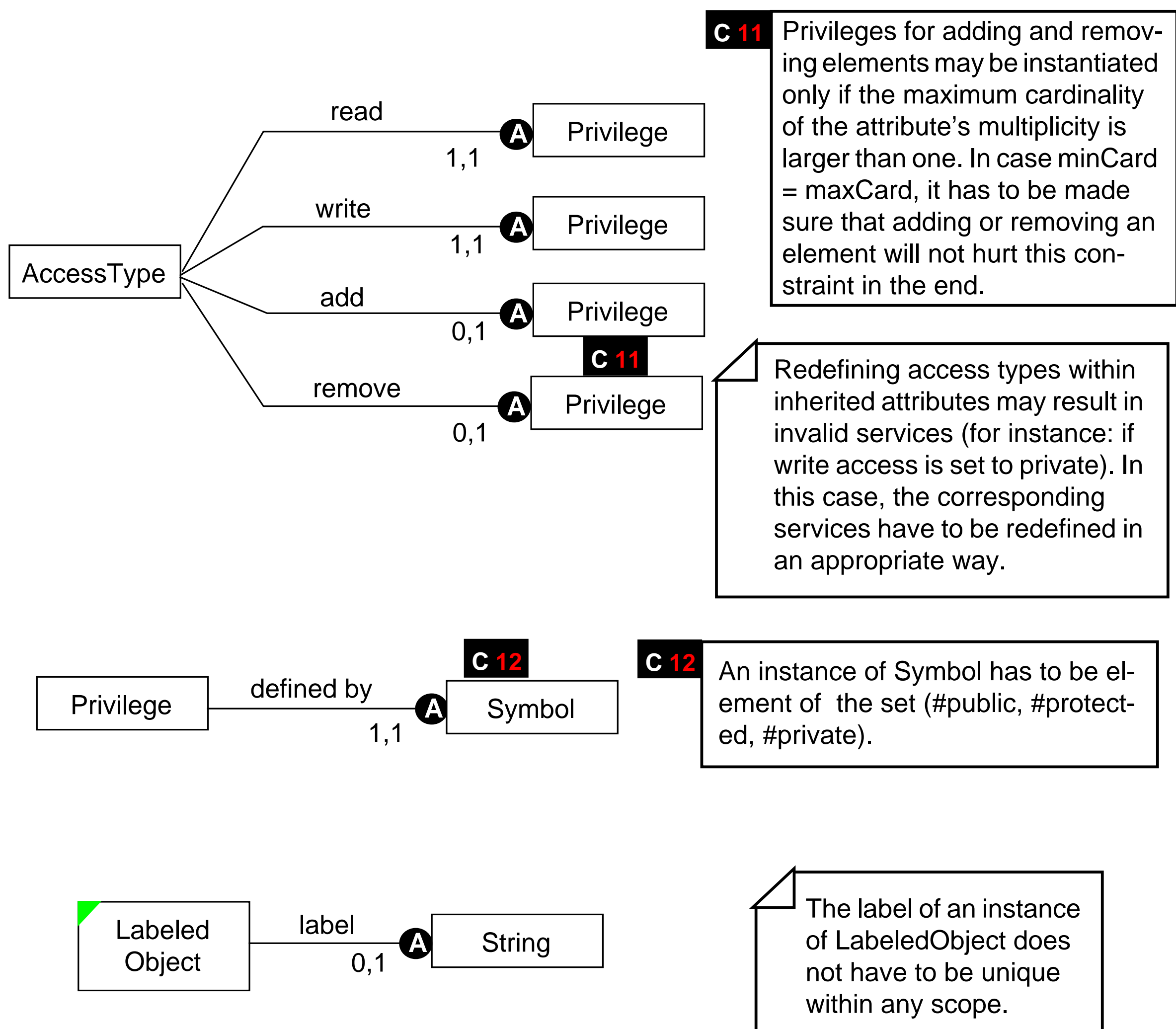


Fig. 20: Access Rights and other Concepts

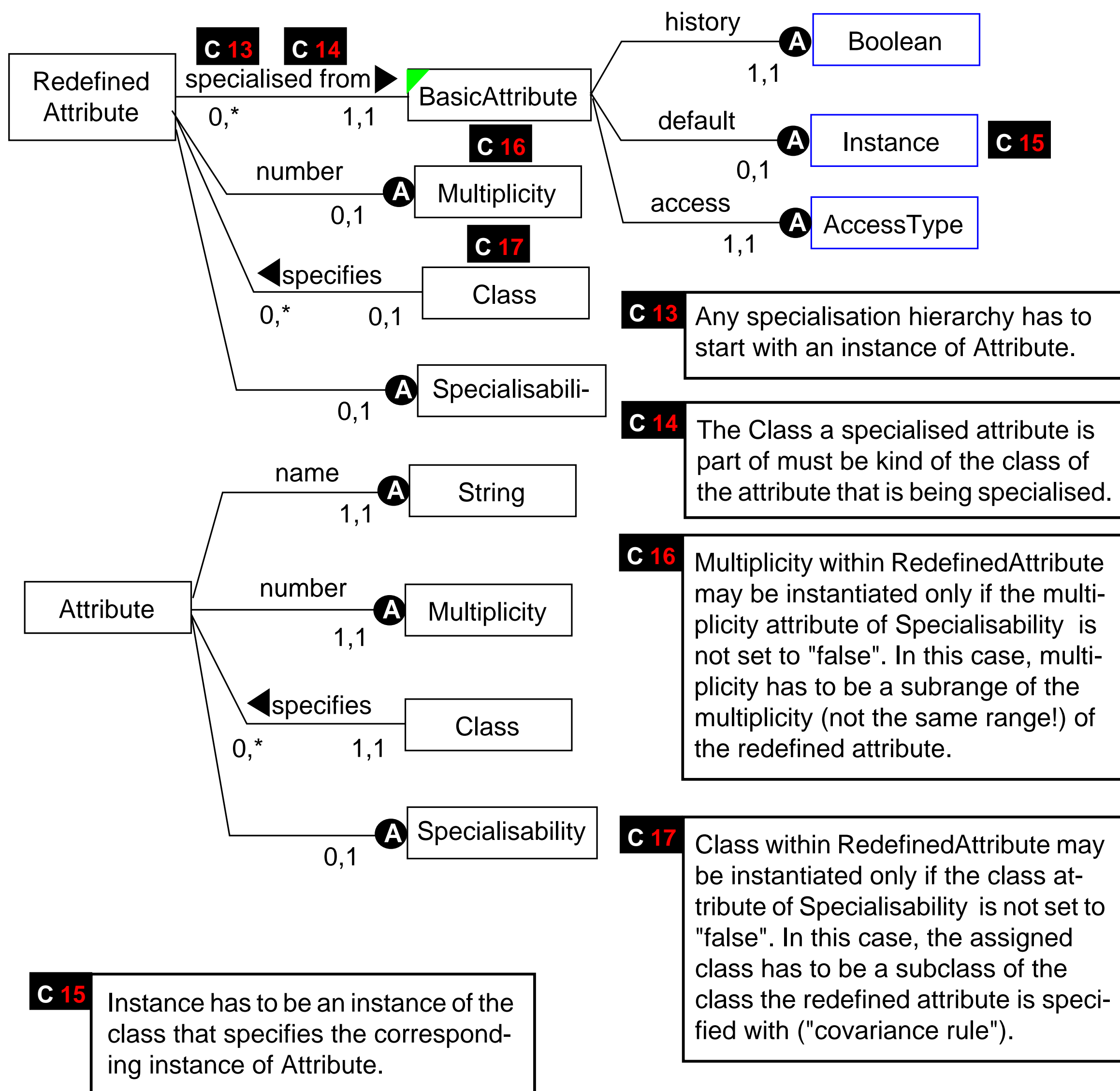


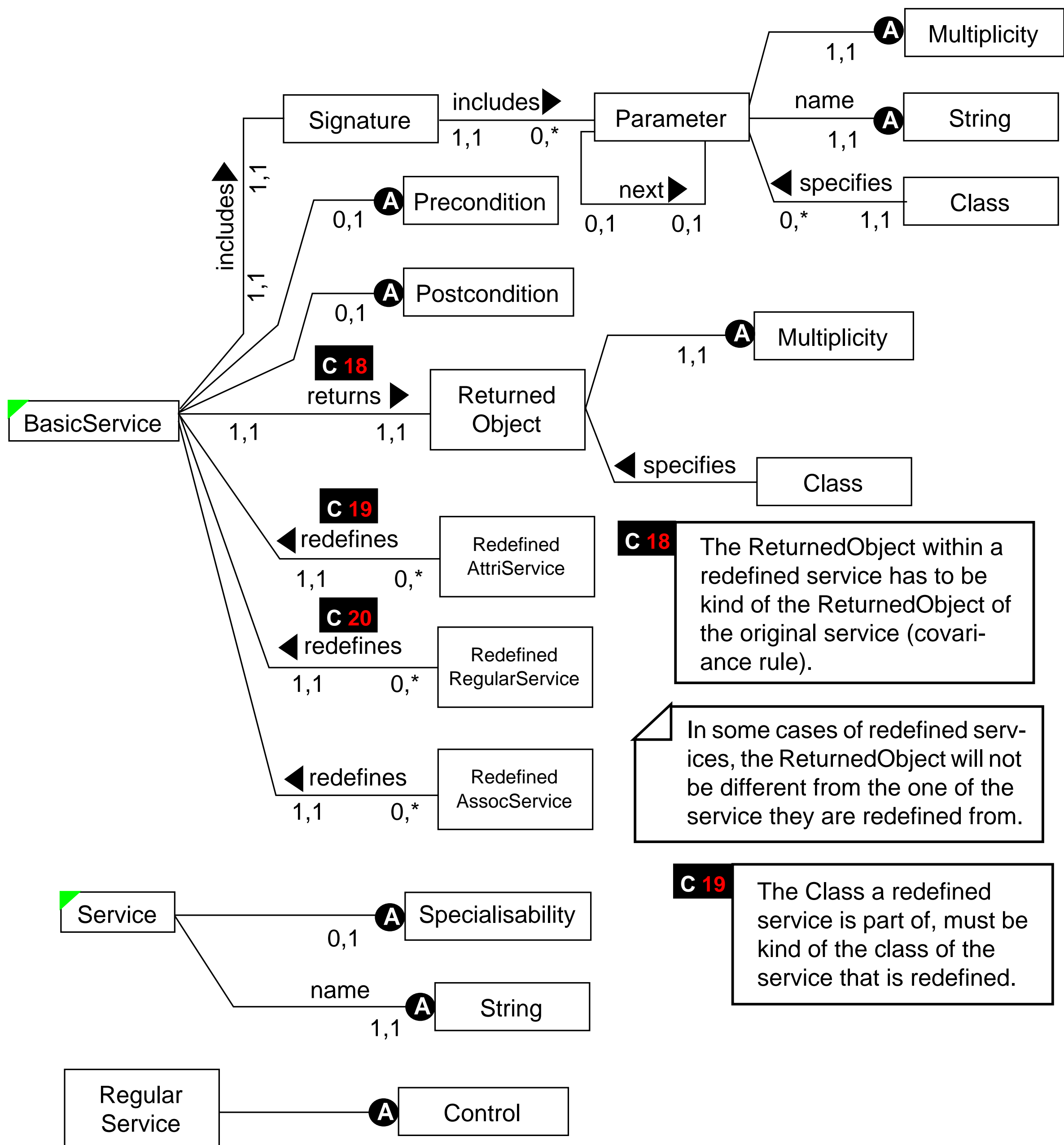
Fig. 21: Attributes (1)

An exception assigned to an inherited service may be redefined. The abstract concept Service was introduced in order to assign a Specialisability attribute. The default values of both, the *class* and the *mulitplicity* attribute of the Specialisability attribute are "true". That is to indicate that classes of inherited parameters and returned objects may be redefined according to the covariance rule or that multiplicities may be redefined respectively.

A redefined service may be specialised from AttriAccessService, AssocAccessService, or RegularService - no matter which BasicService it redefines (see fig. 22). This implies, for instance, that a service inherited from a RegularService can be redefined to become a service to access an associated object (RedefinedAssocService) or an attribute (RedefinedAttriService). Also, a service inherited from an AttriAccessService may be redefined as kind of a RegularService (RedefinedRegularService). As already stated, a service inherited from a AttriAc-

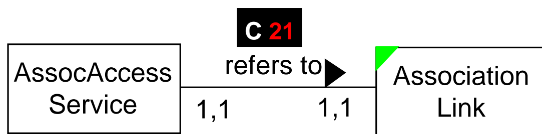
cessService must not be redefined as a RedefinedAttriService. Similar to RedefinedAttriService, a redefined PropagatedService could be introduced. It could be a redefinition of any BasicService except a PropagatedService. However, we assume that there is no need for such a concept. Any type of service allows for the assignment of a Precondition and a Postcondition. However, default access services will usually include an implicit definition of these features. For a DeferredService these assignements are optional, too. Control can be assigned to RegularService and RedefinedRegularServcice only since we assume that it is implicitly defined for default access services or must not be defined for a DeferredService.

As already mentioned above, the different types of service concepts within the metamodel are to support a definition of specialisation on a syntactic level. While such an approach makes it easier to check a model for formal correctness, it may have to be changed when it comes to design a corresponding tool: During the life cycle of a model, model elements may change the concepts they belong to. Within a tool, this would impose the problem to deal with instances that migrate to another class.



C 20 Classes and Multiplicities within Param-eters and ReturnedObject of inherited services may be redefined only if the corresponding attributes of the Special-isability attribute is not set to "false". In this case, the rules for redefining Class (covariance) and Multiplicity apply.

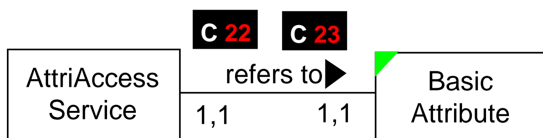
Fig. 22: Services (1)



C 21 An AssocAccessService can be instantiated only if the reference attribute of the corresponding AssociationLink is set to "true".

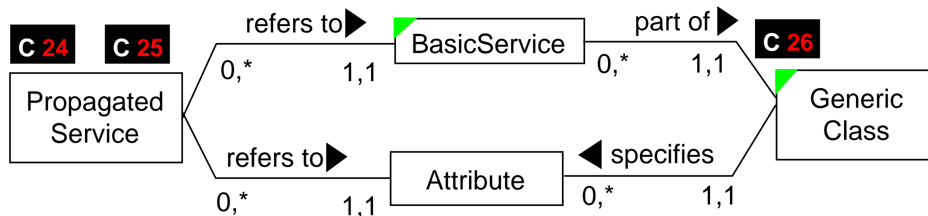
An AssocAccessService has to be part of a Class that is kind of the Class the AssociationLink belongs to.

The Parameters and the ReturnedObject have to correspond to the Class of the associated object.



C 23 An AttriAccessService has to be part of a Class that is kind of the Class the BasicAttribute belongs to.

C 23 The Parameters and the ReturnedObject have to correspond to the Class the BasicAttribute is specified by.



C 24 A PropagatedService has to be part of a GenericClass that is kind of (same as or subclass) the GenericClass the referred Attribute is part of.

C 25 Except for its name and the names of its parameters, the signature of a PropagatedService has to be the same as the one of the referred BasicService. The ReturnedObject has to correspond to the one of the BasicService.

C 26 The BasicService has to be part of the same instance of GenericClass that serves to specify the Attribute.

Fig. 23: Services (2)

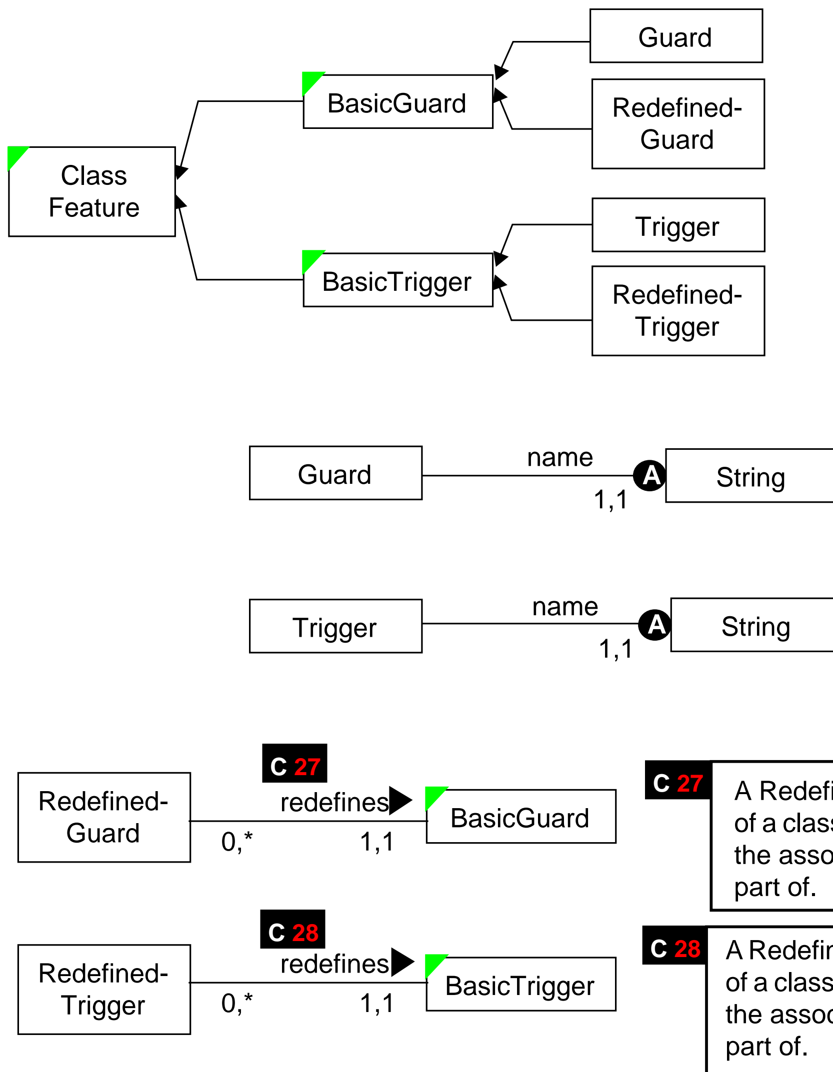


Fig. 24: Triggers and guards

As already stated above, both triggers and guards can be redefined without any restrictions. Notice that this is only due to the fact that there is currently no formal language to specify guards and triggers. Nevertheless, the designer has to beware of arbitrary redefinitions. It is certainly not acceptable for a guard (or a trigger) to contradict a guard (or a trigger) of a superclass. Since guards and triggers will often be transformed into services, it is also important not to hurt the rules for redefining services.

5.2 Associations

Within inherited associations, the class of associated objects and their multiplicities can be redefined (see 4.2). Unfortunately, the differences between the three kinds of associations within MEMO-OML, interaction, aggregation and delegation, do not allow for one common abstraction. Therefore, the corresponding part of the metamodel is large and probably not very readable. AssociationLink is the most general concept. It has four attributes. Specialisability allows

to express whether the class or the multiplicity of a concept (e.g. an attribute or an associated object) may be redefined. If redefinition is prohibited, the corresponding boolean attributes have to be set to "false" (the default is "true"). ContextRole allows to annotate a class that participates in an association. It does not contain any semantics. Nevertheless, if two context roles are used within an association, the names of both roles have to be different (otherwise the concept of a context role would not make much sense). The attribute *history* allows to specify whether any state change of an associated object should be recorded, while the attribute *access* allows to define access rights. Both concepts are used in the same way as they are used for attributes.

Aggregations as well as delegation associations allow to specialise the associated classes. Since the maximum cardinality assigned to a RoleholderLink and an AggregateLink is one, specialising the multiplicity is possible in one case only: from (0,1) to (1,1). Allowing for specialisation of associations results in a more complex and less elegant model (see fig. 25-30). It reflects the rules described for specialising associations in 4.2.

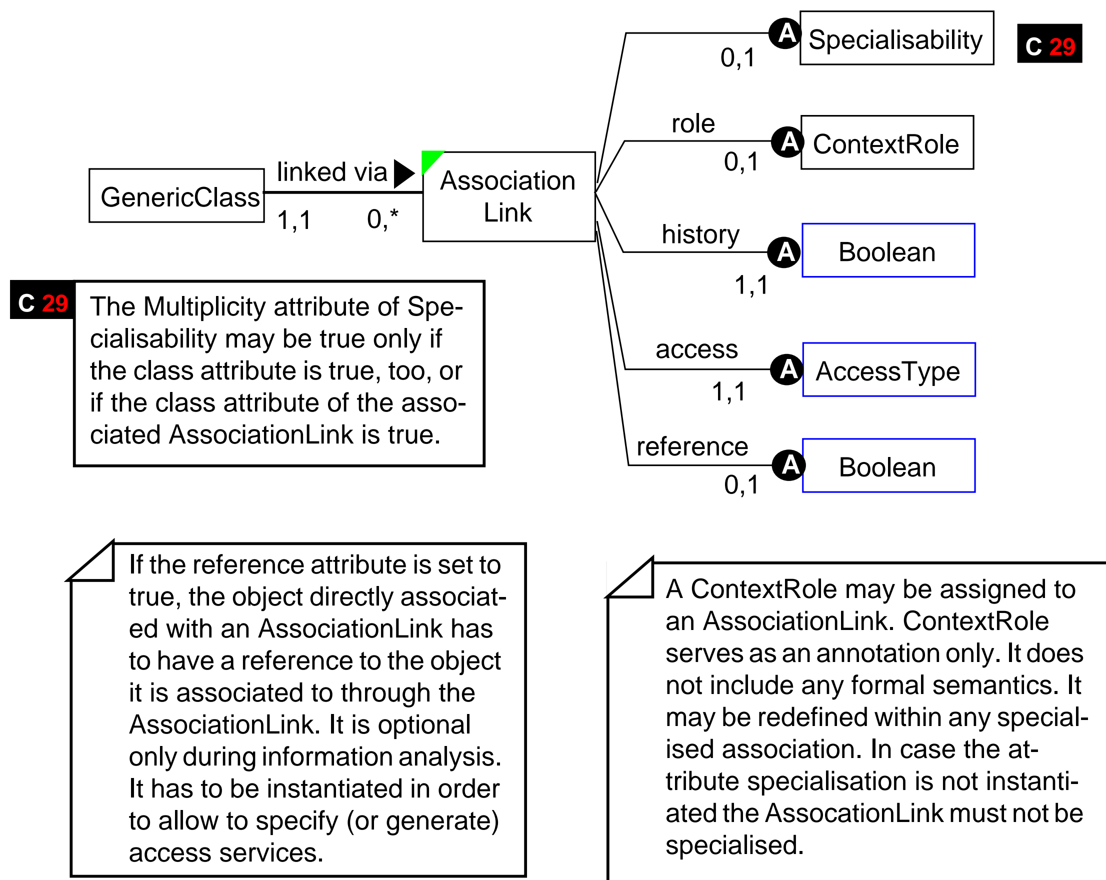


Fig. 25: Associations - Basics

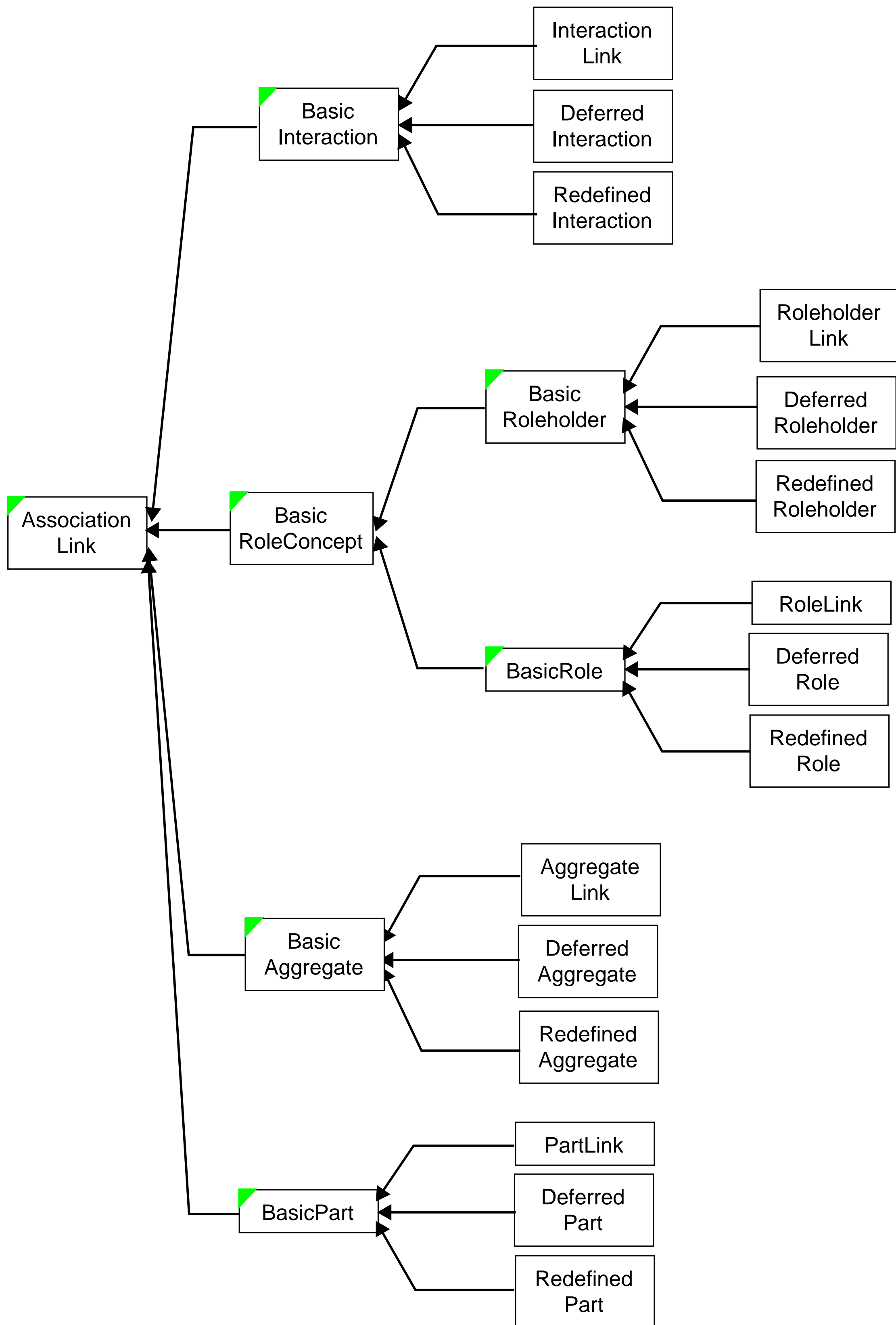


Fig. 26: Associations - Generalisation Hierarchy

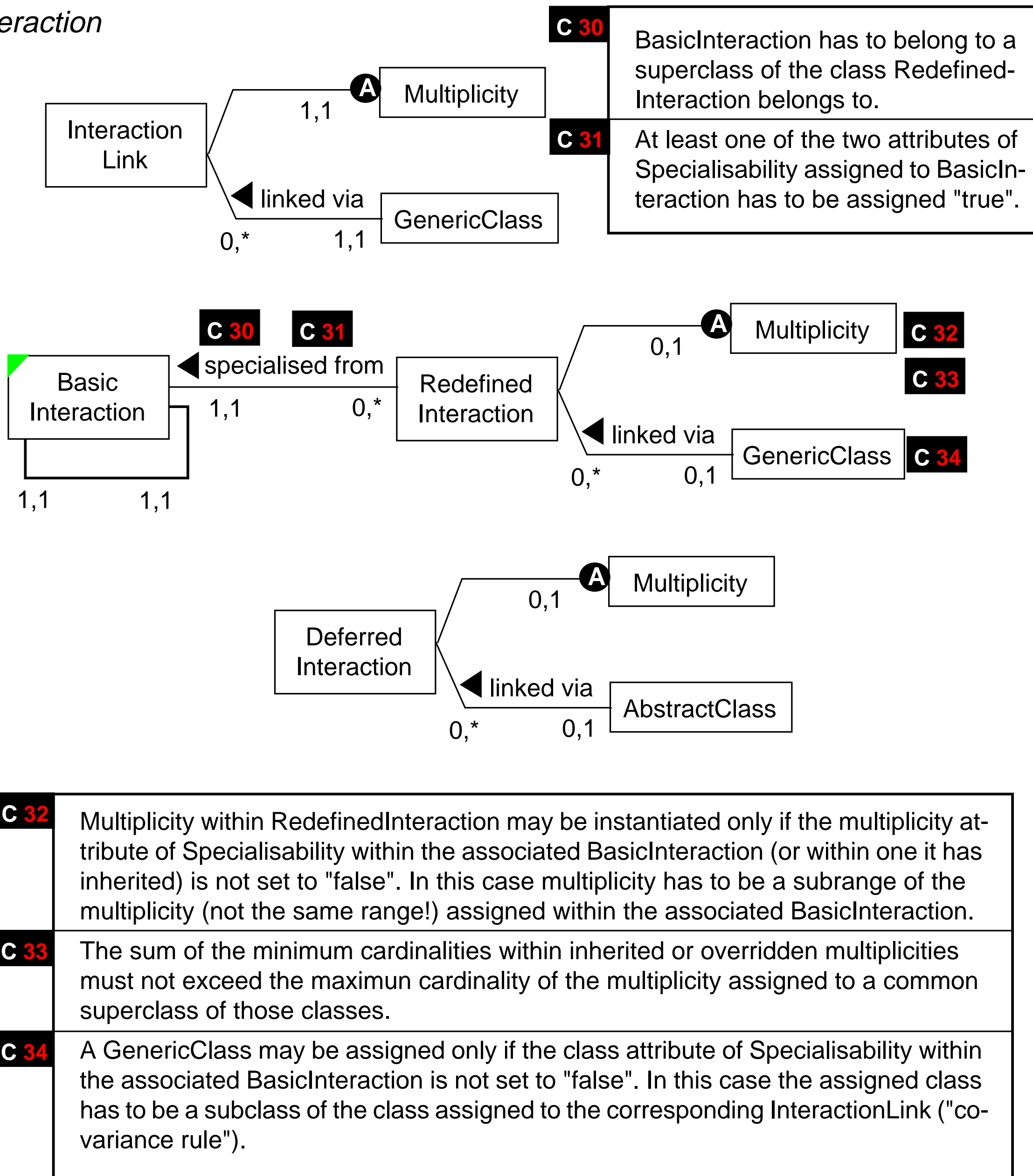


Fig. 27: Interaction Associations

The constraints C1 and C2 in fig. 27 are basically the same for all types of associations. However, since there is no common generalisation for RedefinedInteraction, RedefinedAggregate etc. on a sufficiently specialised level, it is not possible to express general constraints.

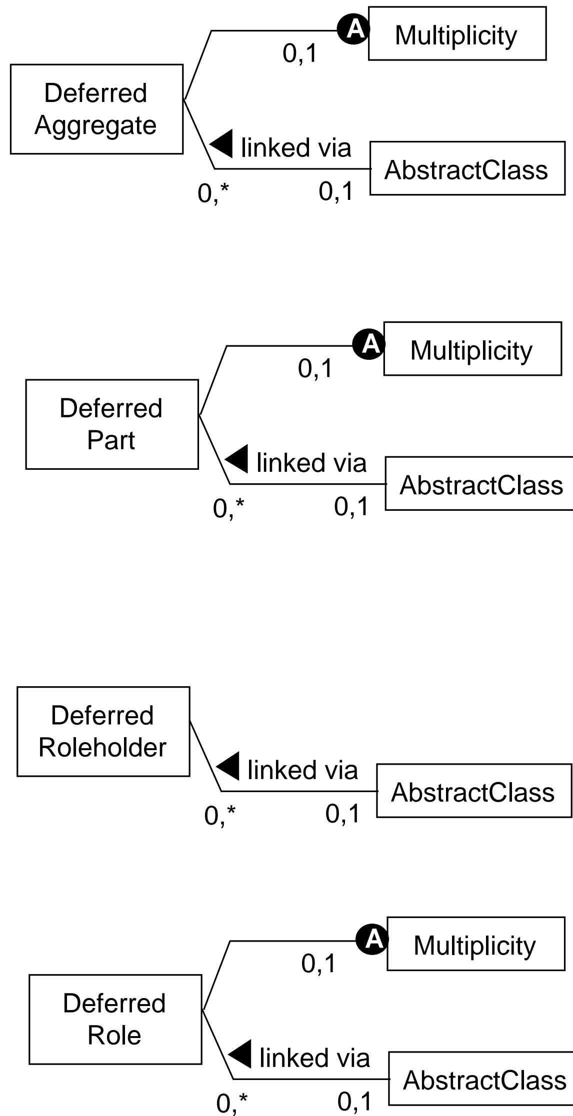
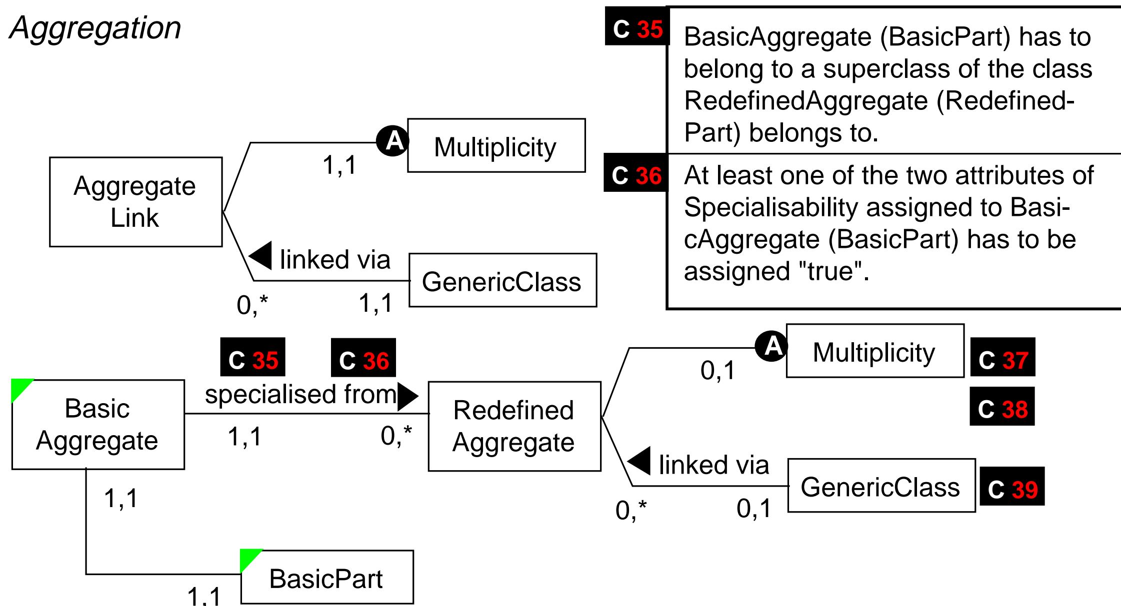


Fig. 28: Deferred Associations

While one could argue that an association is deferred if it is not sufficiently specified (for instance: if one multiplicity is missing), MEMO-OML requires a deferred association to be linked via an abstract class.

Aggregation



C 37	Multiplicity within RedefinedAggregate (RedefinedPart) may be instantiated only if the multiplicity attribute of Specialisability within the associated BasicAggregate (BasicPart) (or within one it has inherited) is not set to "false". In this case multiplicity has to be a subrange of the multiplicity (not the same range!) assigned within the associated BasicPart (BasicAggregate).
C 38	The sum of the minimum cardinalities within inherited or overridden multiplicities must not exceed the maximun cardinality of the multiplicity assigned to a common superclass of those classes.
C 39	A GenericClass may be assigned only if the class attribute of Specialisability within the associated BasicAggregate (BasicPart) is not set to "false". In this case the assigned class has to be a subclass of the class assigned to the corresponding AggregateLink (PartLink) ("covariance rule").

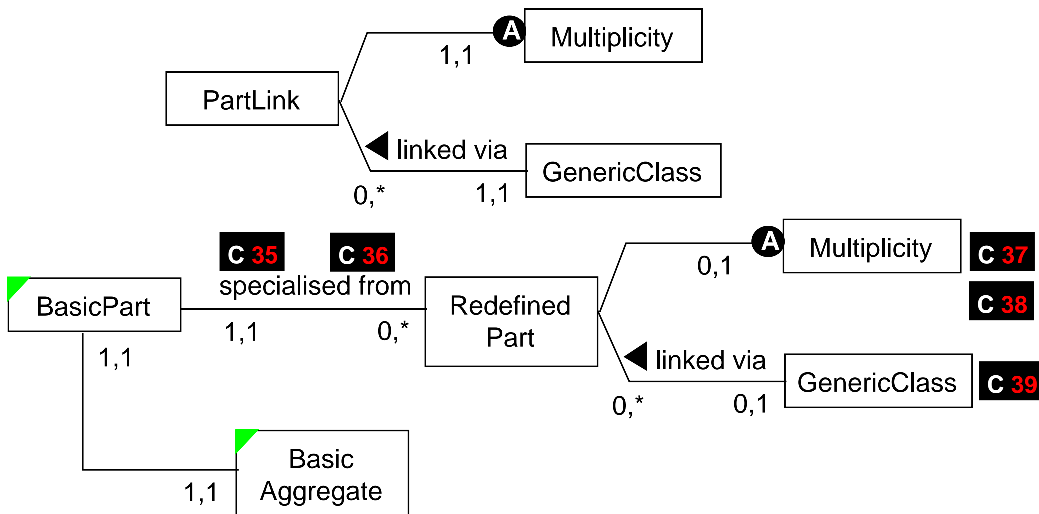
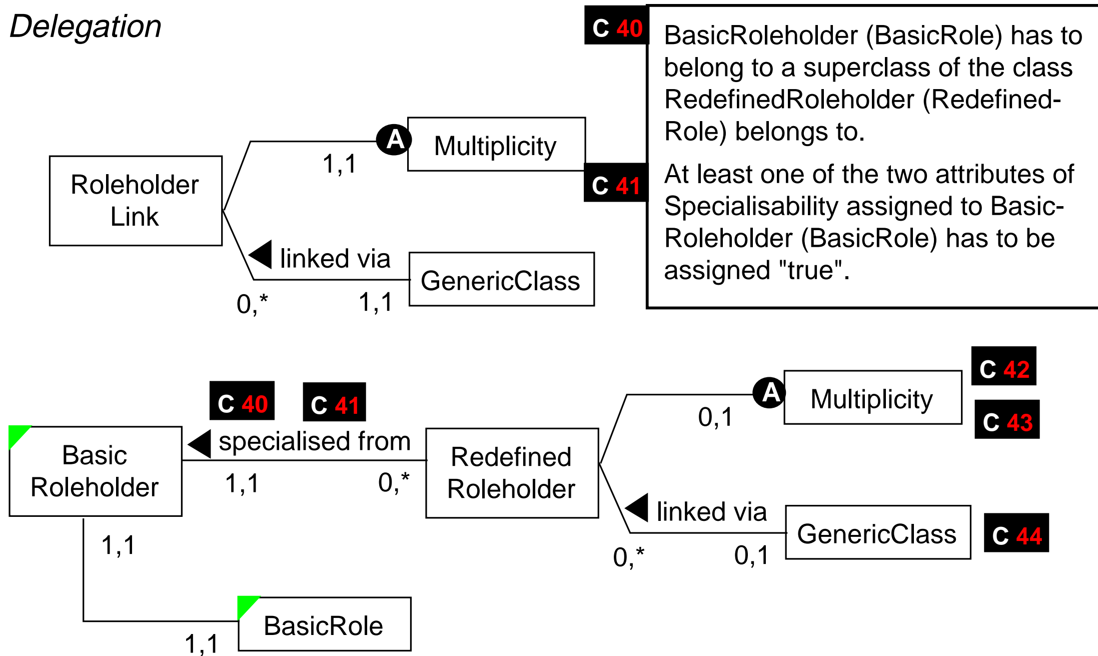


Fig. 29: Aggregations

Delegation



C 42	Multiplicity within RedefinedRoleholder (RedefinedRole) may be instantiated only if the multiplicity attribute of Specialisability within the associated BasicRoleholder (BasicRole) (or within one it has inherited) is not set to "false". In this case multiplicity has to be a sub-range of the multiplicity (not the same range!) assigned within the associated BasicRole (BasicRoleholder).
C 43	The sum of the minimum cardinalities within inherited or redefined multiplicities must not exceed the maximun cardinality of the multiplicity assigned to a common superclass of those classes.
C 44	A GenericClass may be assigned only if the class attribute of Specialisability within the as-sociated BasicRoleholder (BasicRole) is not set to "false". In this case the assigned class has to be a subclass of the class assigned to the corresponding RoleholderLink (RoleLink) ("covariance rule").

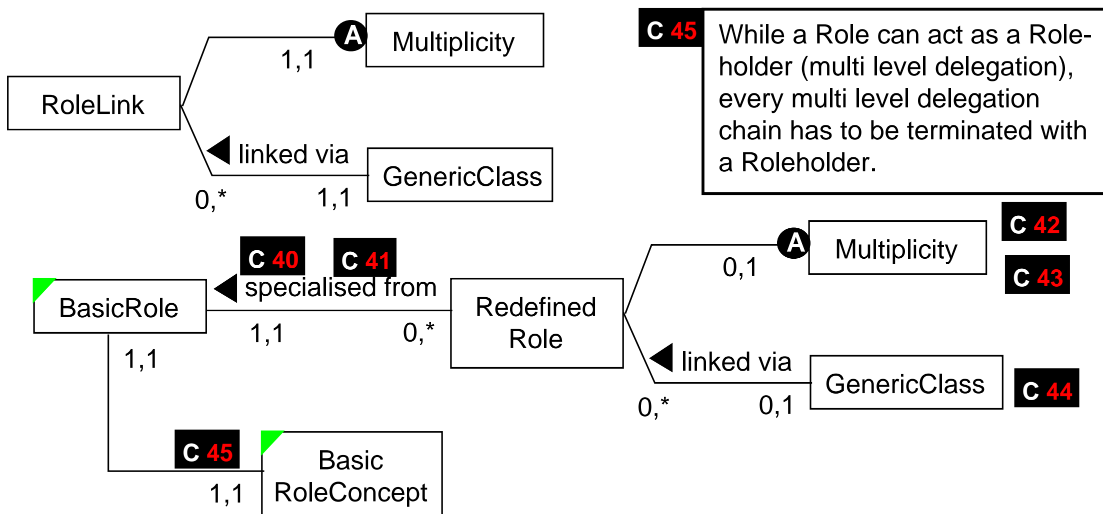


Fig. 30: Delegation

5.3 Metaclasses

While it is arguable whether it makes sense to regard classes as objects within conceptual modelling, this idea can be useful to specify information which is required for system design. For this reason, MEMO-OML offers metaclass as an optional concept. A metaclass is a class with exactly one instance which has to be a class (GenericClass). A class in turn must not have more than one metaclass. We define Metaclass as a concept with two attributes which are specified by MetaAttribute and MetaService. Different from GenericClass, a Metaclass does not have a subclass or a superclass. Therefore (and for other reasons as well), it is not feasible to use the corresponding concepts defined for classes (BasicAttribute or BasicService). Notice that Metaclass does not require a specific name. Instead its name can be composed of "Meta" and the name of its sole instance, like "MetaPerson".

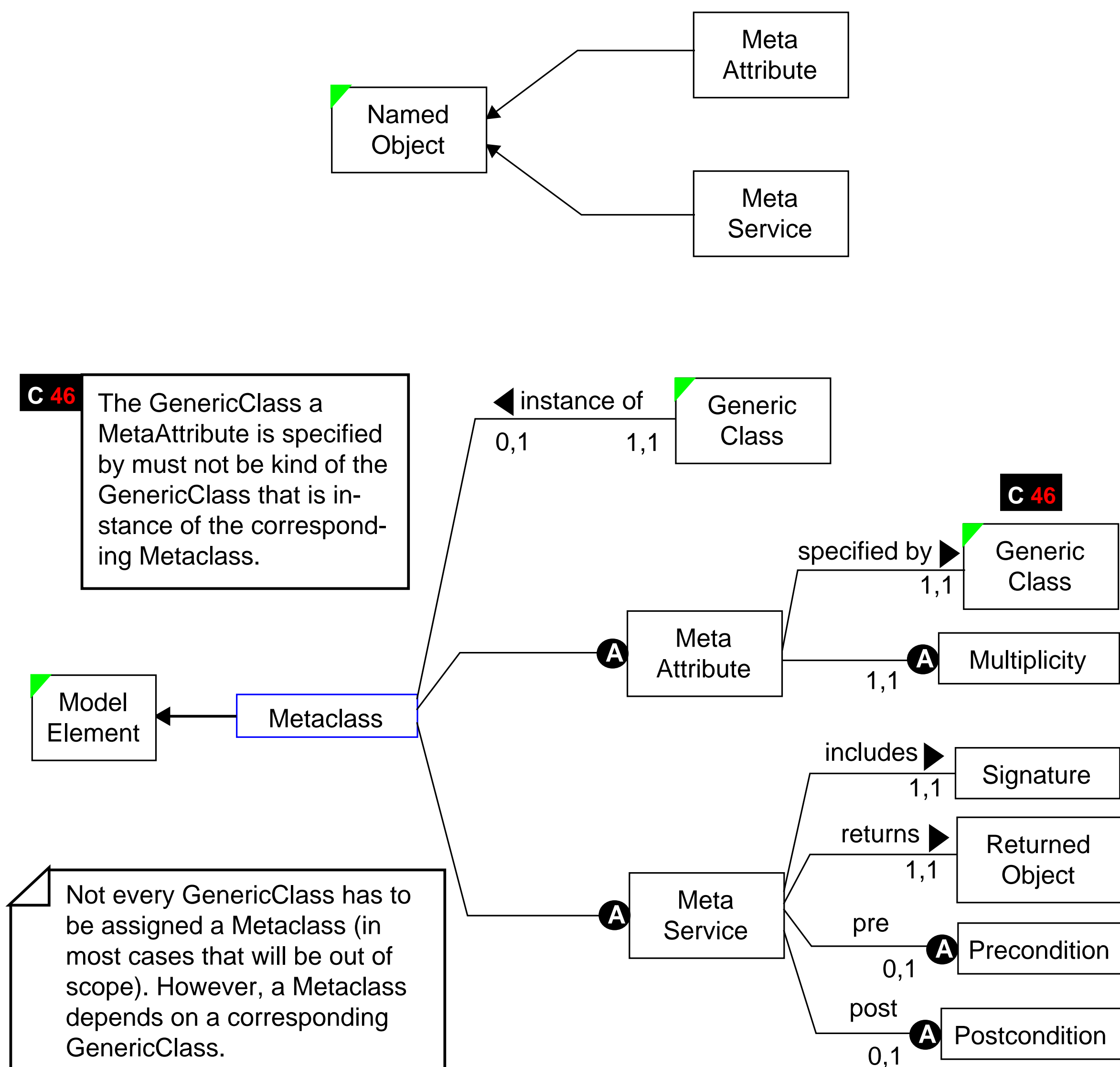


Fig. 31: Metaclass

5.4 Organisation Concepts

The organisation concepts shown in fig. 32, like Category, Protocol, Framework etc., serve to group model elements according to specific puposes. The *visible* attribute of GenericClass allows to express that a class (that is its specification and may be its implementation) should not be visible for further specialisation or even modification. Such a feature can be useful for the description of frameworks. DesignPattern is defined according to the description in 4.3.6. It serves primarily to allow for a structured documentation of clusters and frameworks. Ressource concepts, like Platform, OS or Language, allow to enhance a model with implementation specific information. Notice that these concepts could be specified as classes using MEMO-OML. However, sometimes there is need for a rudimentary description of ressources only. In these cases, it may be regarded as too much of an effort to design an object model for ressources.

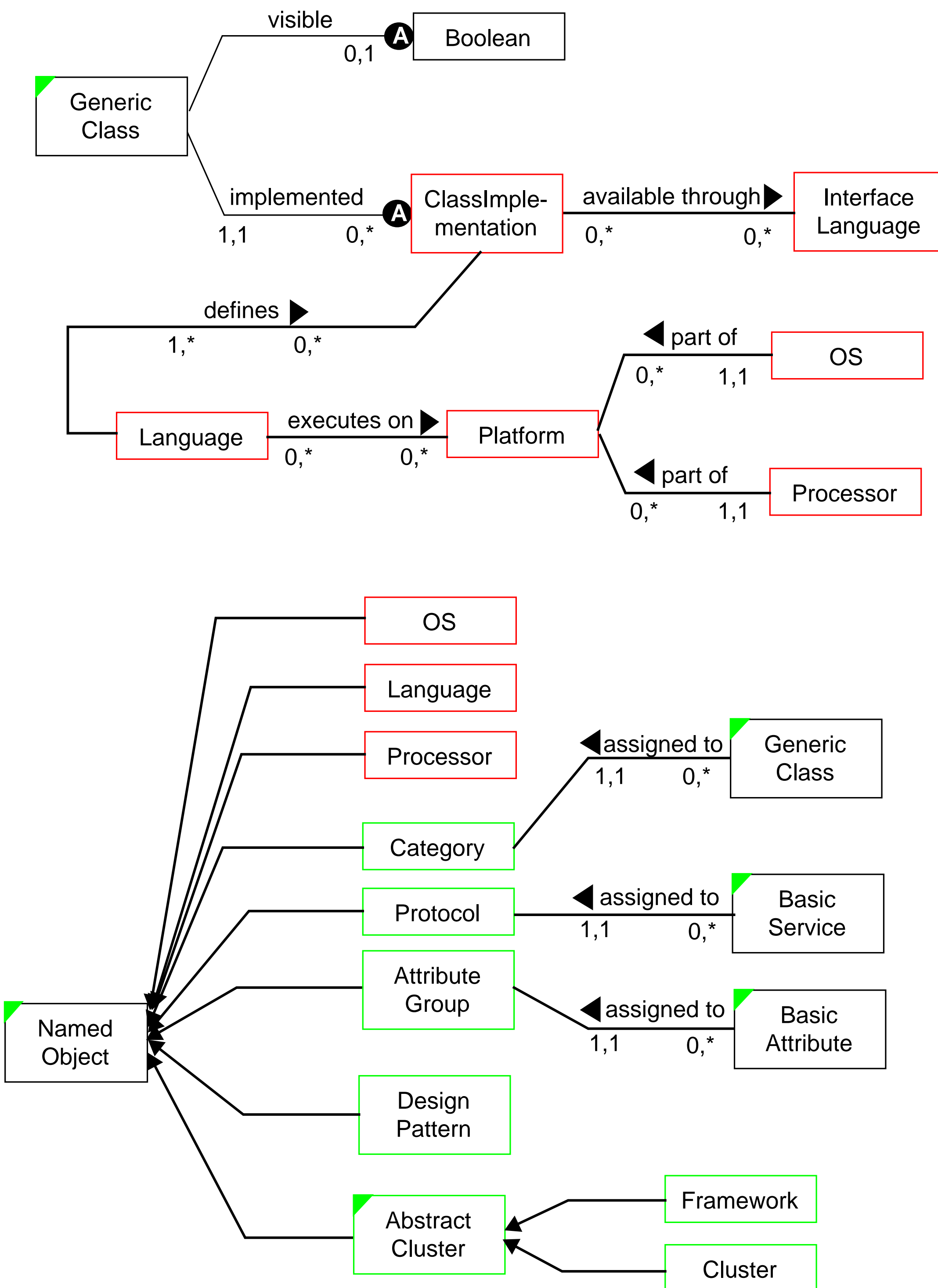


Fig. 32: Modularisation and Architecture

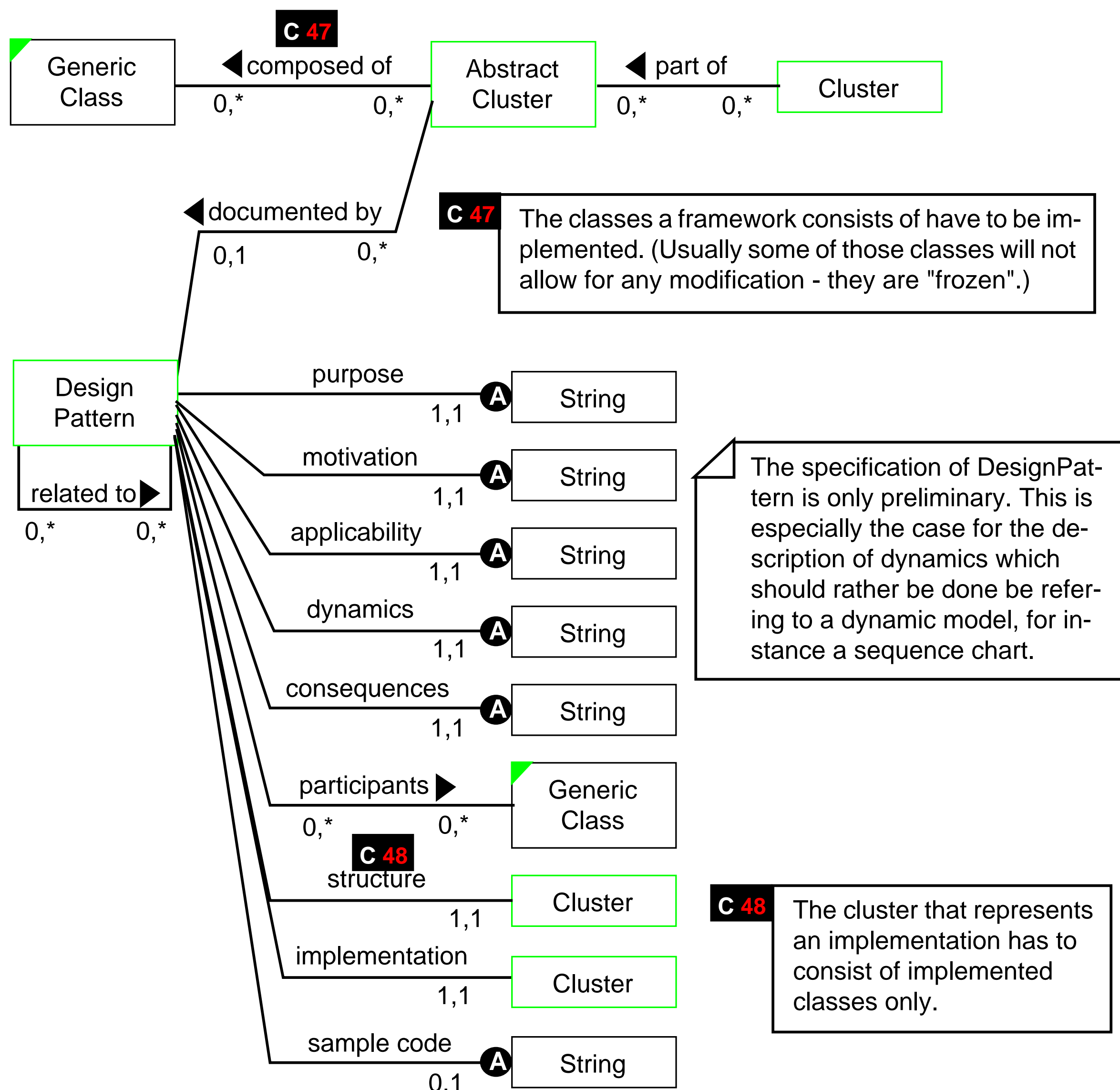


Fig. 33: Modularisation and Architecture

5.5 Stored References to Associated Objects

During most time of conceptual modelling, it is advisable to abstract from the question how references to associated objects are managed. However, in order to prepare for implementation, it can be helpful to add this information. In most implementation level languages, associations are implemented through variables that store references to associated objects. The attribute *reference* within an AssociationLink allows to express whether or not the objects of the corresponding class should store a reference to the object(s) they are linked to through a particular association (see fig. 34). In the final version of an object model, at least one of two associated AssociationLinks should have set reference to "true". This is a "weak" constraint, since

it is possible to store references at other places as well (e.g. in special management objects). Notice that this information is also necessary in order to decide whether or not default access services are available (see 4.3.2).

5.6 Persistence

Every class in a MEMO-OML object model can be assigned the boolean attribute *persistent* that specifies whether or not the instances of this class should be persistent during their lifetime. In order to prepare for indexed access to attributes, an attribute can be assigned the boolean attribute *indexed*. Notice that further specifications may be required to define precisely the content and structure of that information.

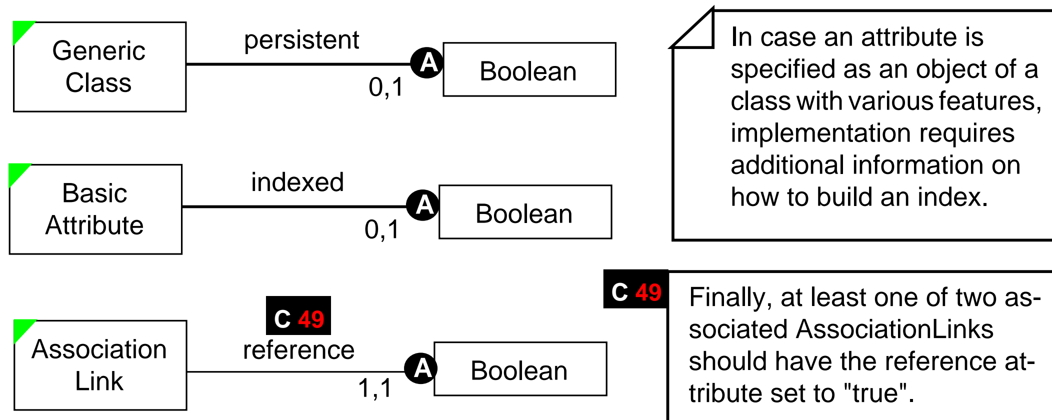


Fig. 34: Implementation-oriented Refinements

5.7 User Interface Concepts

The user interface of a system to be developed is often regarded not to be subject of conceptual modelling, since a conceptual model should focus on features which are essential to the relevant domain. Nevertheless it can be helpful to include information that is relevant for the implementation of a user interface. This is for various reasons:

- *Productivity*: The design and implementation of the user interface consumes a considerable amount of the overall system development time. Enhancing an object model with information concerning the user interface may contribute to faster development cycles by allowing for reuse and code generation.
- *Quality*: The user interface can be a complex part of the entire system. An abstract description improves the chances for a well designed and consistent user interface.

Notice, however, that adding specific information to an object model is hardly efficient for a complete specification of a user interface: The interaction with the classes of a system may vary with the context. An object model, however, does not allow to express certain contexts of interaction with a system. Those contexts can be taken into account within other perspectives, like process or workflow models.

In order to prepare for a rudimentary model of a user interface, it can be helpful to assign a

default user interface to every class in an object model. However, not every class will require an explicit assignment. Often, a user interface can be constructed from the user interfaces previously assigned to (or constructed for) classes of attributes, associated objects, parameters and returned objects. Within an object-oriented information system, the functionality of a user interface will usually be specified by an object model (or more likely: a framework of implemented classes). In other words: User interface concepts are rather an application of an object-oriented modelling language than part of it. Therefore, the preliminary classes we introduce for describing aspects of a user interface will be specified later as an object model using MEMO-OML (see 6). We do need, however, a few enhancements of MEMO-OML in order to allow for UI-specific enhancements of an object model.

The concept GenericClass within the metamodel is modified by adding an optional attribute. The attribute is specified by a class which is described in the user interface object model already mentioned. In the current version, this will be the class AbstractComponent (see fig. 43). Often it will be necessary to present a label with an interaction component. For this purpose, a label can be assigned to any class, attribute or service. In the case of default access services the label can be derived from the accessed attribute or associated object. That label may, however, be overridden. The metamodel defines Label as a concept that can be instantiated with strings for multi-national versions of software.

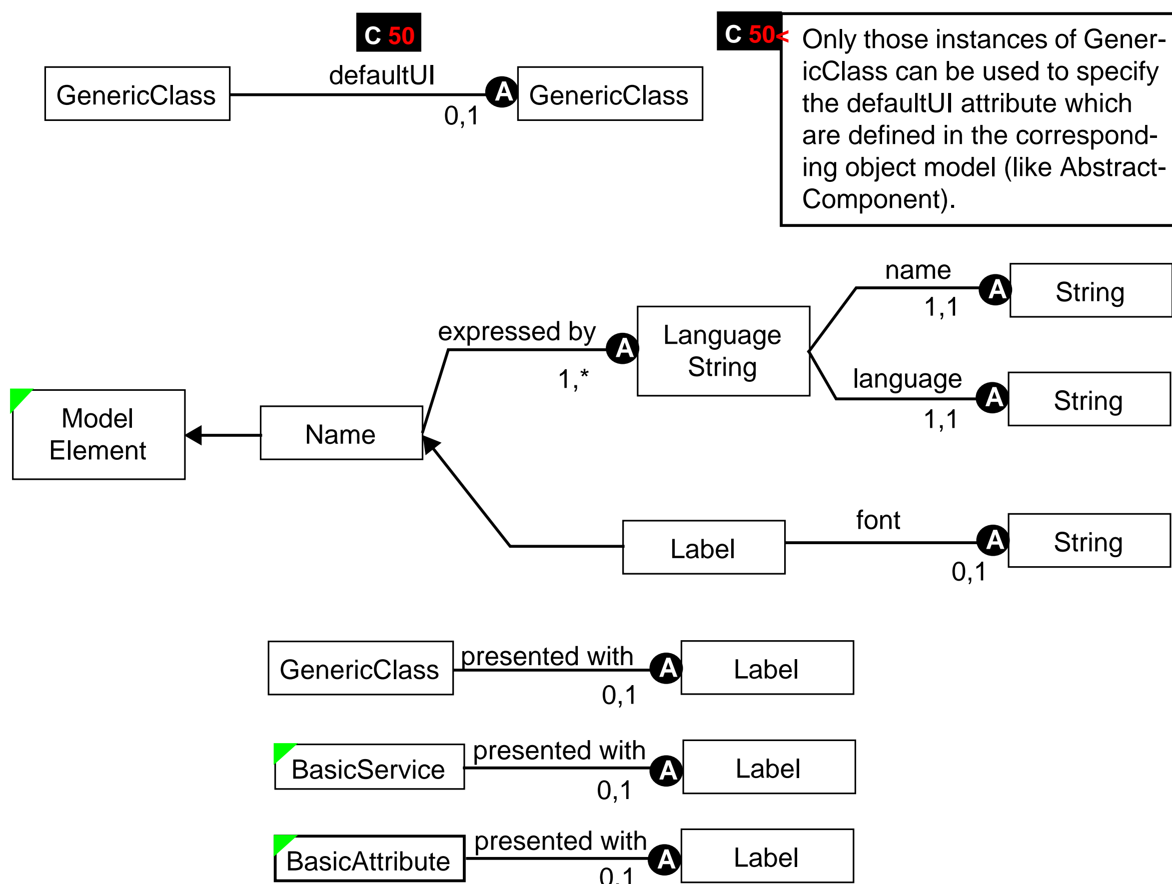


Fig. 35: User Interface Concepts

5.8 The Graphical Notation

Defining a notation for a modelling language is a delicate task. On the one hand, the notation can be exptected to have a pivotal impact on the way people perceive a model. On the other hand, there is lack of research on user perceptions and preferences concerning graphical representations of information ([Fra97]). In this situation, the definition of a graphical notation has to be preliminary. While it should be based on plausible assumptions, feedback from future users may result in revised versions. The current version of the MEMO-OML notation is primarily based our own preferences. We also tried to take into account the following requirements (see [Rum96a]) some of which are hard to judge:

- no overloading of notation elements
- clear distinction between different elements of the notation
- clear correspondence between concepts and elements of the notation
- convenient to draw by hand
- easy to memorize
- elements that are used often should be to render quickly
- appropriate for single colour printers and fax machines

The notation is similar to the one we have already used for rendering the metamodel. Notice that the colours used within the notation are optional. The description of the notation introduced below is not complete in the sense that it includes explicitely all possible elements of the notation. Introducing a graphical notation of redefined associations is motivated by the fact that redefinitions are often required. Therefore, a specific notation fosters the convenience and the consistence of using MEMO-OML. Notice, however, that in the case of multiple inheritance the arrow indicating a redefined association may be ambique: There may be different associations of the same kind (but with different interpretations) a class has inherited from different superclasses. In this case an additional constraint has to be used to specify the association that has been redefined.

Due to its graphical nature, the notation is not described in a formal way. Therefore the use of the symbols listed below allows for - and recommends - a certain degree of flexibility. This is especially the case for the layout of models, e.g. for connecting symbols via arcs or lines. The example arcs/lines shown below are not to indicate that any arc/line in a particular model has to be drawn in a corresponding way.

5.8.1 Naming Conventions

Strings used for naming modelling elements must be in line with naming conventions defined for these elements. We will, for instance, suggest class names to begin with an upper case letter. In contrast, names used for attributes or services should begin with a lower case letter. If precise specification of naming conventions are available, they can be assigned to the attribute *naming* of any appropriate instance of *MetaConcept* within the meta-metamodel.

The textual designators used within the graphical representation of an object model are usually not of crucial importance: Often additional specifications will be documented outside a diagram - either on paper or within a tool. Nevertheless, we will provide a precise definition of the designators, some of which are optional. For this purpose we use the terminal symbols in-

troduced for the description of metamodels (see 2.1) and extend them with additional symbols to be used with MEMO-OML only. Since the decision for the formal language to specify constraints, guards, etc. has not been made yet, <expression> remains undefined at this stage. Also, the definition does not include any semantic constraints. The non-terminal symbols used in the description of the graphical symbols are typed in boldface.

Access Privileges (corresponds to AccessType and Privilege in the metamodel)

<privilege> ::= 'public' | 'protected' | 'private'

<getAccess> ::= 'Get: ' <privilege>

<putAccess> ::= 'Put: ' <privilege>

<addAccess> ::= 'Add: ' <privilege>

<removeAccess> ::= 'Remove: ' <privilege>

Associations

<rolename> ::= <lowerString>

Classes

<classname> ::= <upperString>

<maxLabel> ::= 'max. size: ' | 'max. number: ' | ...

<maxInstances> ::= '(' <maxLabel> <PositiveInteger> ')'

<classlabel> ::= <classname> [<maxInstances>]

Constraints/Comments

<constraintkey> ::= 'C' <number>

<commentkey> ::= <number>

Attributes

<defaultValue> ::= ', default: ' <String>

<fixedSymbol> ::= 'F'

<redefinedSymbol> ::= 'R'

<deferredSymbol> ::= 'O'

<ordered> ::= 'ordered' | 'sorted'

<attributeName> ::= <lowerString>

<regularAttributeSpec> ::= <attributeName> <classname> [<multiplicity>] [<defaultValue>] [<ordered>] [<getAccess>] [<putAccess>] [<addAccess>] [<removeAccess>]

<deferredAttributeSpec> ::= <deferredSymbol> <regularAttributeSpec>

<redefinedAttributeSpec> ::= <redefinedSymbol> <regularAttributeSpec>

<fixedAttributeSpec> ::= <fixedSymbol> <regularAttributeSpec> "attribute that must not be redefined"

<attributeSpecList> ::= [{<regularAttributeSpec> <LineFeed>}] [{<deferredAttributeSpec> <LineFeed>}] [{<redefinedAttributeSpec> <LineFeed>}]

Services

In case the maximum cardinality within the multiplicity of a parameter or a returned object is less than 2, it is optional to explicitly specify it.

<serviceBaseName> ::= <lowerString>

<paramSpec> ::= <lowerString> ' [' <classname> [<multiplicity>] ']'

<smalltalkStyleSig> ::= <serviceBaseName> [': ' <paramSpec>] [{<lowerString>': ' <paramSpec>}] ' ->' <classname> [' - ' <multiplicity>]

<cStyleSig> ::= <serviceBaseName> ['(' <paramSpec> [{', ' <paramSpec>}]')'] ' ->' <classname> [' - ' <multiplicity>]

<regularSignatureSpec> ::= <smalltalkStyleSig> | <cStyleSig> [<privilege>]

<deferredServiceSpec> ::= <deferredSymbol> <regularSignatureSpec>

<redefinedServiceSpec> ::= <redefinedSymbol> <regularSignatureSpec>

<fixedServiceSpec> ::= <fixedSymbol> <regularServiceSpec> "service that must not be redefined"

<propagatedServiceSpec> := <regularSignatureSpec> 'prop. from: ' <attributeName>

<serviceSpecList> ::= [{<regularServiceSpec> <LineFeed>}] [{<deferredServiceSpec> <LineFeed>}] [{<redefinedServiceSpec> <LineFeed>}]

Guards and Trigger

<guardSpec> ::= <lowerString> ['(' <expression> ')']

<redefinedGuardSpec> ::= <redefinedSymbol> <guardSpec>

<guardSpecList> := [{<guardSpec>}] [{redefinedGuardSpec}]

<triggerSpec> := <lowerString> ['(' <expression> ')]

<redefinedTriggerSpec> ::= <redefinedSymbol> <TriggerSpec>

<triggerSpecList> := [{<TriggerSpec>}] [{redefinedTriggerSpec}]

Redefined Associations

<redefinedID> ::= 'S ' <number>

<classRedefinitionSpec> ::= 'redefines: ' <classname> ' in: ' <classname> <designator> <classname> ' with: ' <classname>

<multiRedefinitionSpec> ::= 'redefines multi for: ' <classname> ' in: ' <classname> <designator> <classname>

Management and Organisation

<frameworkName> := <upperString>

<clusterName> := <upperString>

<patternName> := <upperString>

<platformName> := <String>

Any name used for a class feature must be unique within all class features of the same kind in the scope a class. Additionally, it should be taken into account that default access services are related to certain attributes or associated objects. In order to illustrate this relationship, it is a good idea that the names of the access services correspond to the name of the attribute. For this purpose, we suggest the following conventions for naming default access services, using a notation that is inspired by the Bachus-Naur form but less rigid. The definition of attributeName and serviceBaseName are provided above. The names of access services for a particular attribute with the name 'example' would be constructed as follows (restricted to signatures specified in the Smalltalk style):

<aBase> ::= 'example'

<anUpper> ::= 'Example'

<aParamSpec> ::= <paramSpec>, with the name of the attribute's class used as a terminal symbol for <classname>. <lowerString> could be represented by 'a', <classname>, in this case with the name of the class the attribute is part of as a replacement for <classname>.

The colon indicates that a parameter has to be provided. Using the plural form of the attribute name can be considered as a preliminary default that is not always appropriate since the attribute name itself may already be in the plural. Also, at this point, it is not necessary to consider possible linguistic rules which would allow to generate a plural form.

Multiplicity	get	put	remove Element	add Element
0,1	<aBase>	<aBase> ':' aParamSpec		
0,*	plural of <aBase>	plural of <aBase> ':' <aParamSpec>	'remove' <anUpper> ':' <aParamSpec>	'add' <anUpper>: <aParamSpec>

In case the elements of an attribute are ordered, the additional access services would be named as follows:

get last	get first	add after Element
'last' <anUpper>	'first' <anUpper>	'addAfter' <anUpper> ':' <aParamSpec>

add before Element	get after Element	get before Element
'addBefore' <anUpper> ':' <aParamSpec>	'after' <anUpper> ':' <aParamSpec>	'before' <anUpper> ':' <aParamSpec>

In case services have to be specified in a different style, the construction rules have to be adapted (for instance to the C-style described above.

Example:

Attribute	Multiplicity	get	put	remove Element	add Element
lastName	0,1	*lastName*	*lastName: aString [String]*		
language	0,*	*languages*	*languages: aString [String - 0,*]*	*removeLanguage: aString [String]*	*addLanguage: aString [String]*

Such a naming convention makes sense only if the signatures of services other than default access services do not interfere with it. Generating names for default access services that provide access to associated objects is more difficult. At first sight - and this will do in many cases - it seems to be sufficient to apply a convention that directly corresponds to the one used for services that access attributes: Instead of the attribute's name you would take the name of the associated class. For instance: Accessing the department an employee is assigned to would be accomplished by services like *department* or *department:*. However, in a case where a class is associated more than one time with a particular other class (see example in fig. 36), this convention is obviously not sufficient.

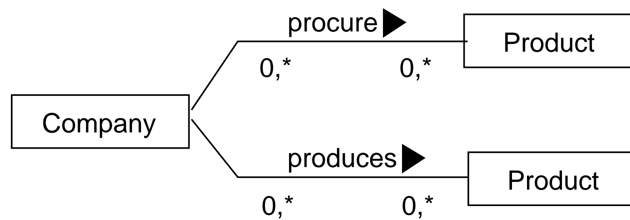
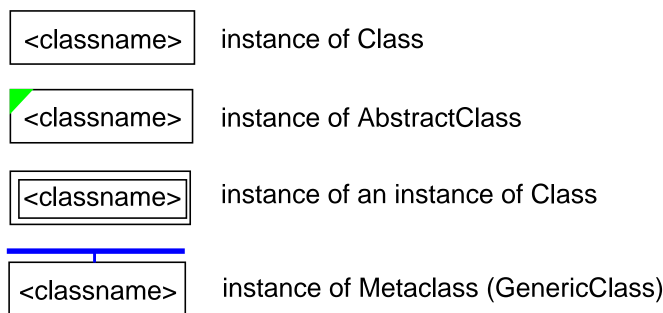


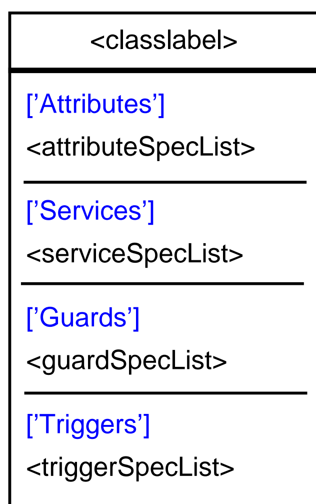
Fig. 36: Different Associations with the same Class.

In those cases, the readability of a model requires the use of - otherwise optional - labels for the associations. Those labels can then be used to compose unique service names - like "procuredProducts". For the purpose of a language description it is not essential to care about the maintenance of a model. This is different with an object model for a tool that serves to manage models designed in this language. For such an object model it is a good idea to enforce a certain naming convention in order to avoid misleading names.

5.8.2 Graphical Symbols

The following figures render the graphical symbols to be used with MEMO-OML.





Class Specification

It is recommended to use labels for the different categories of class features, and to draw lines between them.

Fig. 37: MEMO-OML Notation (1)

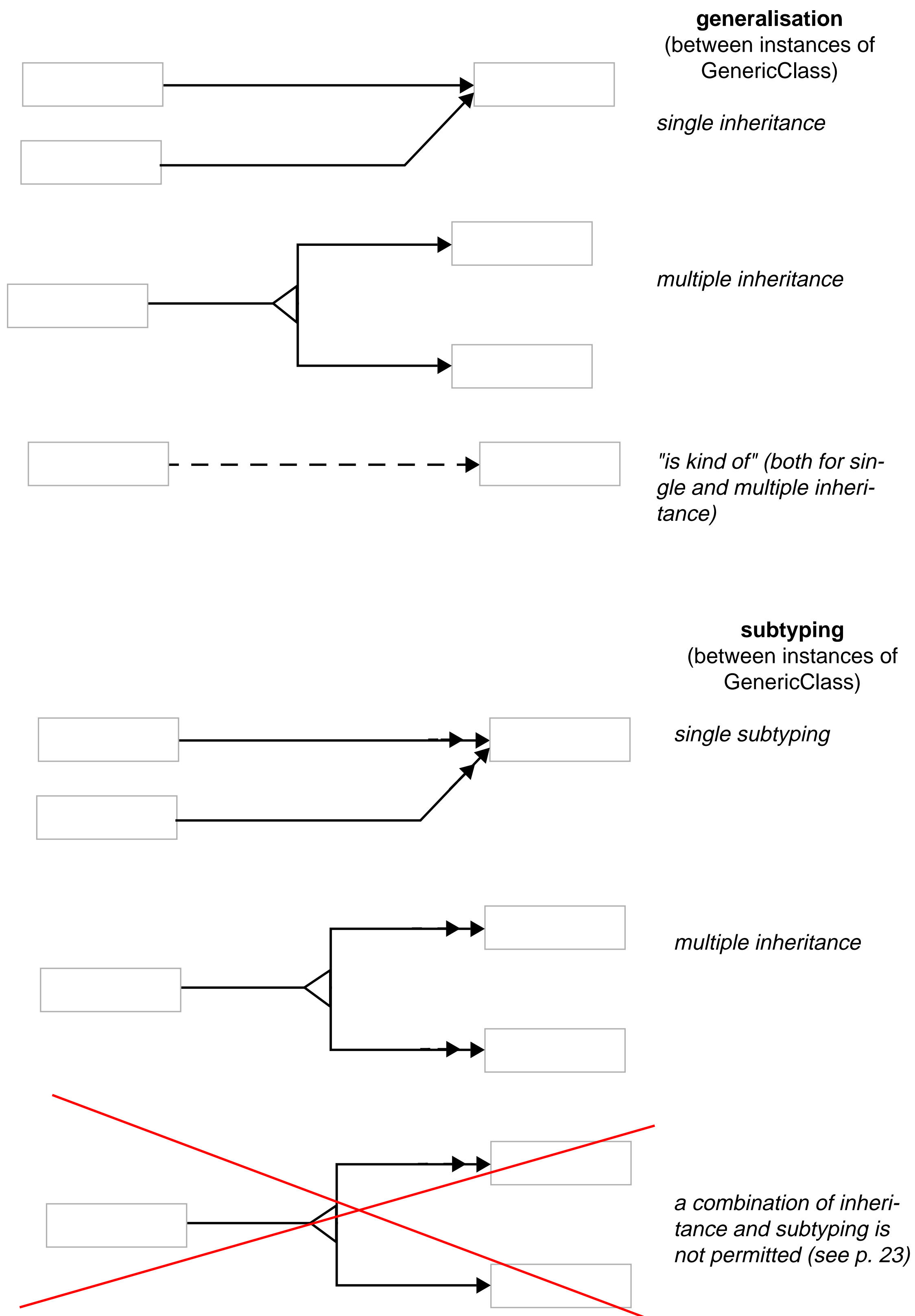
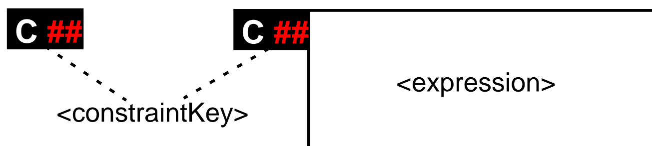
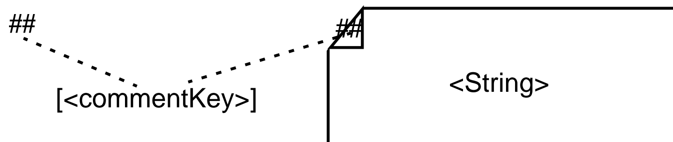


Fig. 38: MEMO-OML Notation (2)

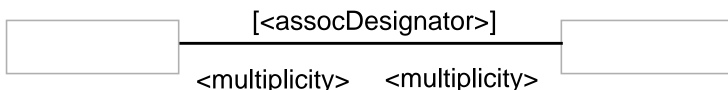


Constraint
(instance of Constraint, n serves as a unique reference within a model). The use of colour for the identifier is optional.

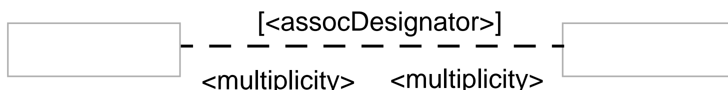


Comment
(instance of Comment)

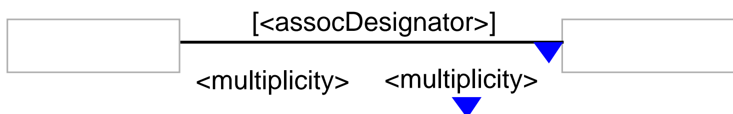
Associations (1)



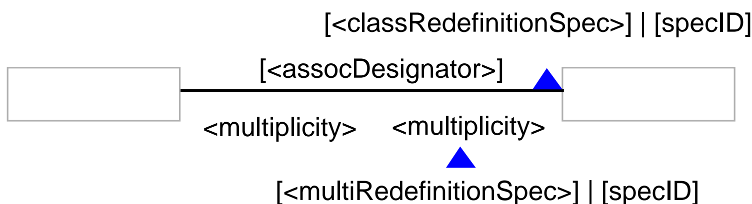
Interaction Association



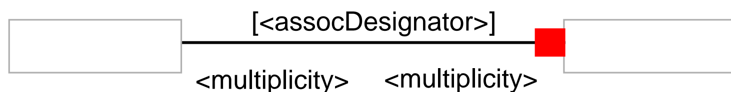
deferred Association



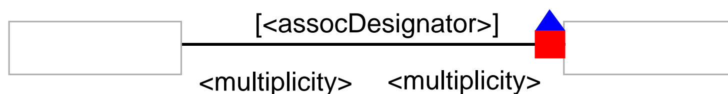
association link that must not be redefined (to be used explicitly for class and multiplicity)



redefined Association (to be used explicitly for class and multiplicity)



Reference



Combination of symbols

Fig. 39: MEMO-OML Notation (3)

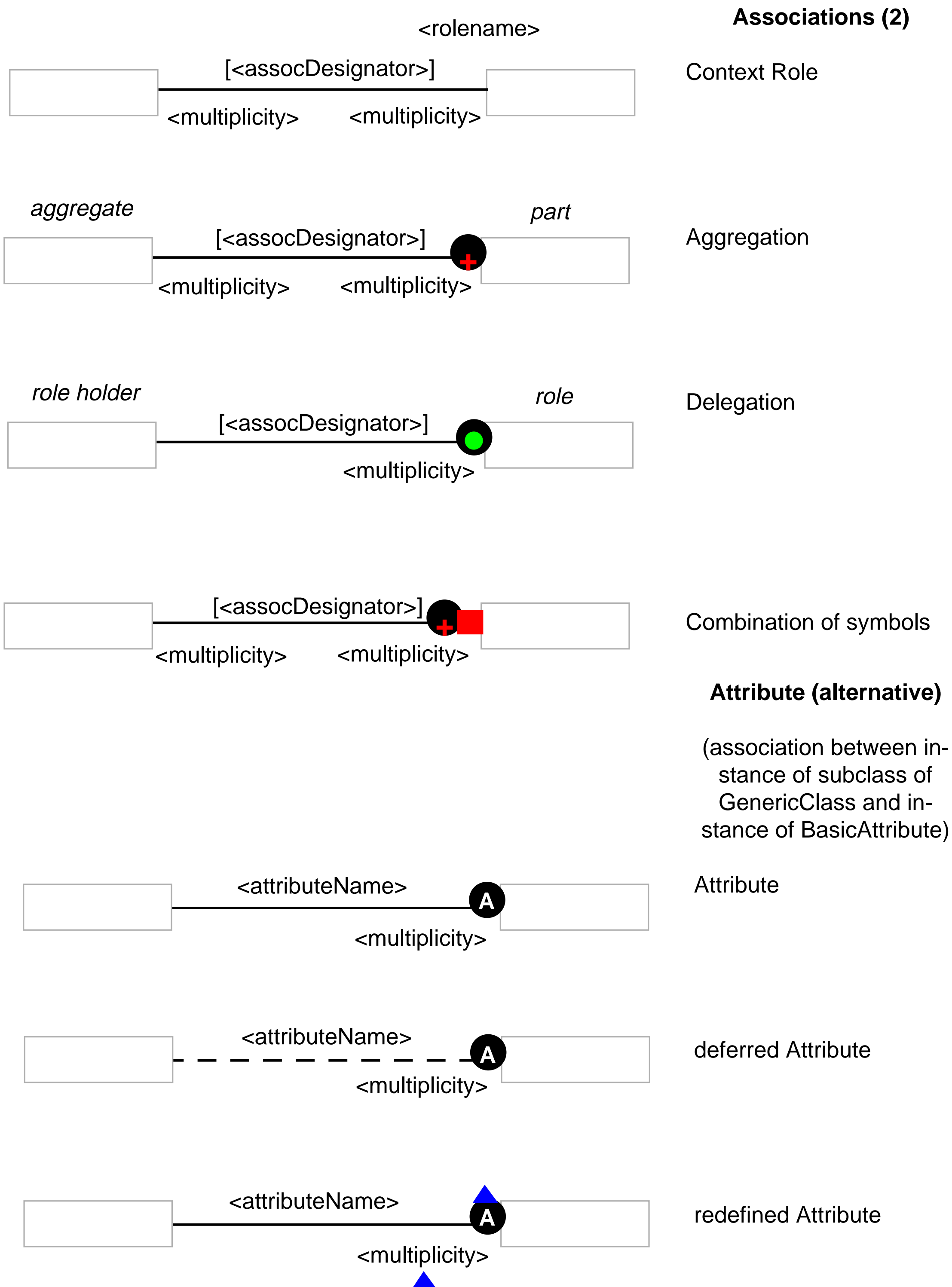


Fig. 40: MEMO-OML Notation (5)

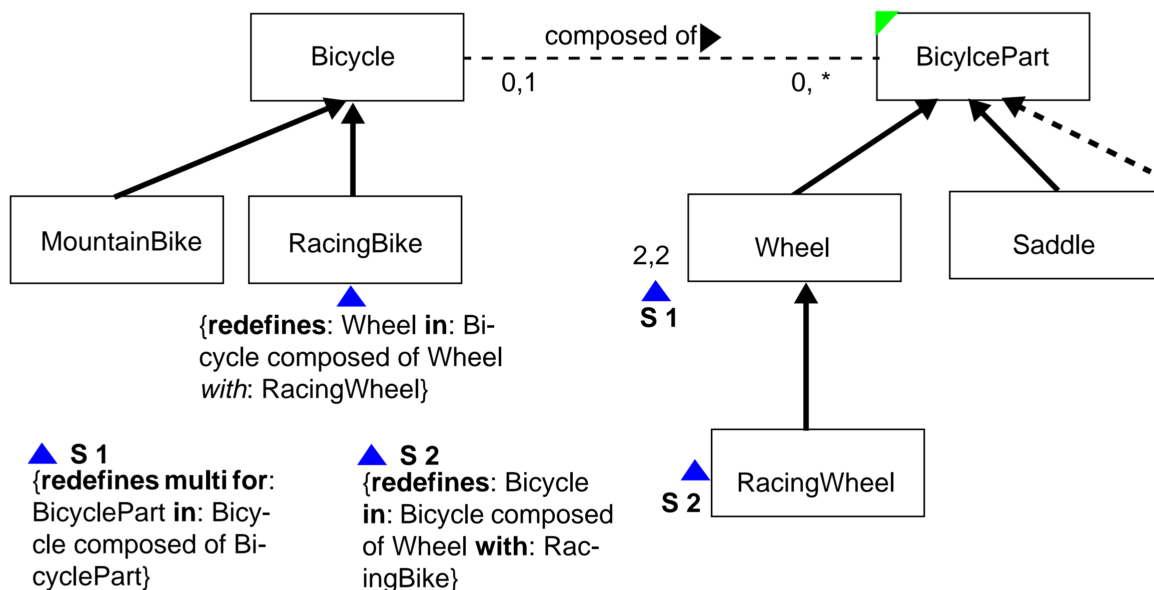


Fig. 41: Example for the notation to express redefined associations.

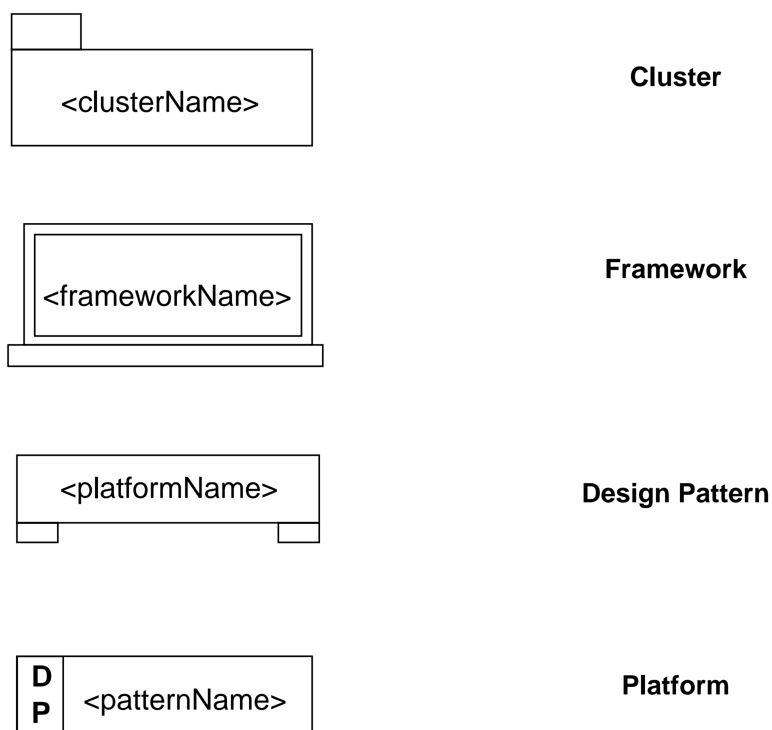


Fig. 42: MEMO-OML notation (5)

6. User Interface Classes

The preliminary object model that describes the classes to be used for the optional specification of default user interfaces is based on the well known paradigm of today's window-oriented graphical user interfaces. BasicComponent is an abstract class that can be specialised in an expandable number of interaction components, like TextField, ListBox, PushButton, etc. A detailed specification of those components will be subject of a future report. In general, a BasicComponent can be assigned to any class. However, some classes may require additional specification in order to allow for such an assignment: A BasicComponent allows to present instances of certain classes (usually of String) only. In case a class does not fit that requirement, it could be assigned a service that delivers a representation of an instance in an appropriate way (like: *asString*). As an alternative, such a class could be implicitly assigned a UIComposite. A UIComposite consists of other UIComposites and/or of BasicComponents (see fig. 43). A UIComposite assigned to a particular class can be derived recursively from the AbstractComponents (see fig. 43) that are assigned to the classes used to specify that particular class' attributes, as well as the parameters and returned objects of its services.

It makes a difference whether one or more instances of a class are to be presented at one time. Therefore each class can be assigned an additional BasicComponent (like ListBox) that allows to interact with multiple instances. Any class within an object model can be assigned a class that serves to describe user interaction elements. Additionally, any class, attribute or service is assigned a label it is to be presented with. Notice that, despite the identical notation, fig. 43 does not render a part of the MEMO-OML metamodel. Instead, it shows an object model defined in MEMO-OML.

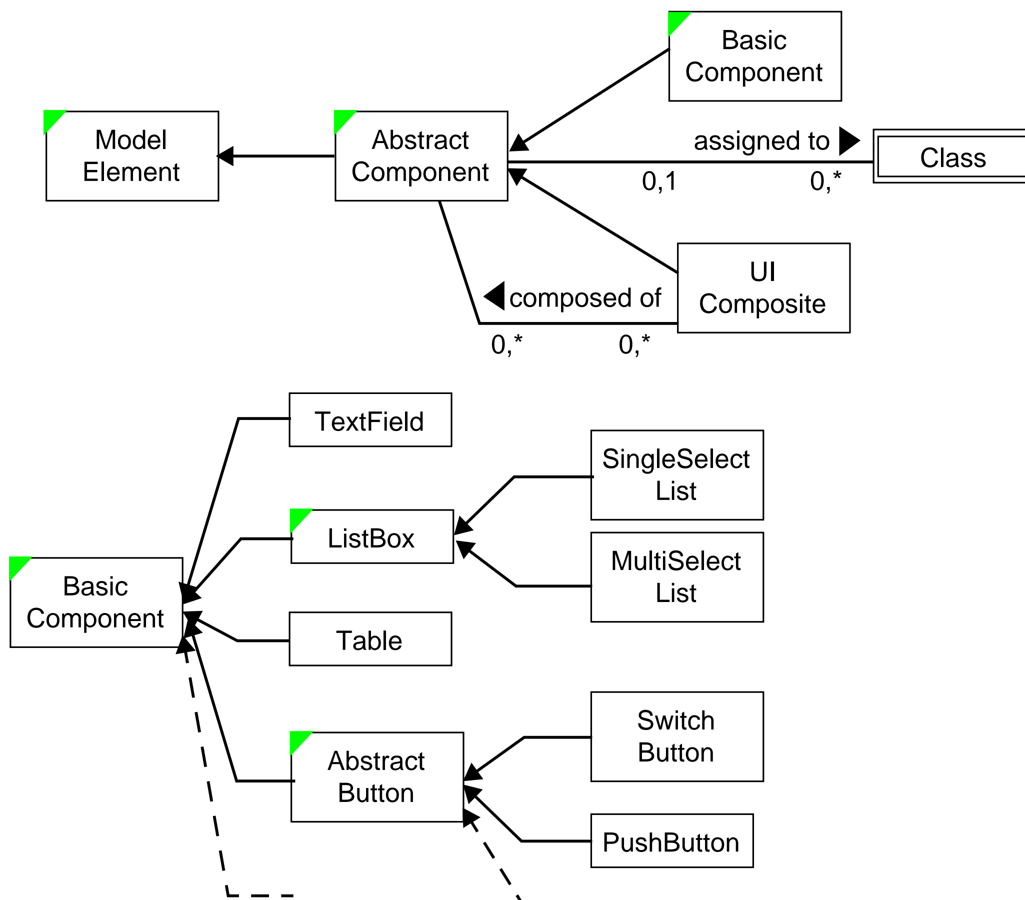


Fig. 43: Object Model of User Interface

7. Future Work

We do not consider the specification of an object-oriented modelling language to be an end in itself. It is to serve various purposes and users. In other words: It is an instrument for different tasks used by people with different backgrounds. For this reason only, it is not advisable to freeze a given modelling language without evaluating it against important requirements. Such an evaluation recommends to have the language used by different users and refine it step by step according to the feedback you get. With this intention in mind, MEMO-OML will be used within software development projects at the institute and for teaching object-oriented modelling to students. One essential goal of future refinements is to fill the semantic gaps of the current version, especially to introduce a specification language for constraints, pre- and postconditions, trigger etc.

In the end, it is not satisfactory to use an object-oriented modelling language without the support of a tool. Without a tool, it is very likely that a model will become inconsistent. Browsing, searching and code generation are additional reasons to use a tool. Also, the construction and maintenance of graphical representation can be more convenient if it is supported by a power-

ful tool. Therefore, we intend to build a new version of the Object Model Designer (OMD) which will be part of the integrated tool environment MEMO Center ([Fra97]). In order to be compatible with standard modelling languages, we plan to provide a transformation of MEMO-OML models into other representation - which, however, may lead to a lack of semantics.

References

[Car87] Cardelli, L: Basic polymorphic typechecking. In: Science of Computer Programming, 8(2), April 1987, pp. 147-172

[Cas95] Castagna, G.: Covariance and Contravariance: Conflict without a Cause. In: ACM Transactions on Programming Languages and Systems. Vol. 17, No. 3, 1995, pp. 431-447

[CaBa97] Cattell, R.G.G.; Barry, D.; Bartels, D. et al. (Ed.): The Object Database Standard - ODMG 2.0. San Francisco: Morgan Kaufmann 1997

[CoPo95] Cotter, S.; Potel, M.: Inside taligent technology. Reading/Mass. et al.: Addison-Wesley 1995

[Doy79] Doyle, J: A Truth-Maintenance System. In: Artificial Intelligence 12, 1979, pp. 231-272

[EbFr95] Ebert, J.; Franzke, A.: A Declarative Approach to Graph Based Modeling. In: Mayr, E.; Schmidt, G.; Tinhofer, G. (Eds.): Graphtheoretic Concepts in Computer Science. Berlin, Heidelberg etc.: Springer (LNCS 903) 1995, pp. 38-50

[EbSü97] Ebert, J.; Süttenbach, R.; Uhe, I.: Meta-CASE in Practice: a Case for KOGGE. In Olive, A., Pastor, J. A. (Eds.): Advanced Information Systems Engineering, Proceedings of the 9th International Conference, CAiSE'97. Berlin, Heidelberg etc.: Springer (LNCS 1250) 1997, pp. 203-216

[EbWi96] Ebert, J; Winter, A; Dahm, P.: Graph Based Modelling and Implementation with EER/GRAL. Fachberichte Informatik, Universität Koblenz-Landau, Heft 11, 1996

[Ern97] Ernst, J.: Introduction to CDIF. 1997 (http://www.cdif.org/)

[FiHe96] Firesmith, D.; Henderson-Sellers, B.; Graham, I.; Page-Jones, M.: OPEN Modeling Language (OML). Reference Manual. Version 1.0. 8 December 1996. http://www.csse.swin.edu.au/OPEN/comn.html

[Fra97] Frank, U.: Enriching Object-Oriented Methods with Domain Specific Knowledge: Outline of a Method for Enterprise Modelling. Arbeitsberichte des Instituts fuer Wirtschaftsinformatik, Nr. 4, Koblenz 1997

[Fra98a] Object-Oriented Modelling Languages: State of the Art and Open Research Questions. In: Schader, M.; Korthaus, A. (Ed.): The Unified Modeling Language. Technical Aspects and Applications. Heidelberg, New York: Physica 1998, pp. 14-31

[Fra98b] Frank, U.: The MEMO Meta-Metamodel. Arbeitsberichte des Instituts für Wirtschaftsinformatk, Nr. 9, Koblenz 1997

[FrHa97] Frank, U.; Halter, S.: Enhancing Object-Oriented Software Development with Delegation. Arbeitsberichte des Instituts für Wirtschaftsinformatk, No. 2, Koblenz 1997

[Frz97] Franzke, A.: GRAL 2.0: A Reference Manual. Fachberichte Informatik, Universität Koblenz-Landau, 1997

[GaHe95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Reading/Mass. et al.: Addison-Wesley 1995

[GoSt90] Goldstein, R.C.; Storey, V.C.: Some Findings on the Intuitiveness of Entity Relationship Constructs. In: Lochovsky, F.H. (Ed.): Entity Relationship Approach to Database Design and Query. Amsterdam: Elsevier 1990

[Hit95] Hitchman, S.: Practitioner Perceptions on the Use of some Semantic Concepts in the Entity Relationship Model In: European Journal of Information Systems, Vol. 4, 1995, pp. 31-40

[Joh92] Johnson, R.E.: Documenting frameworks using patterns. In: A Framework for Network Protocol Software. In: Proceedings of the OOPSLA'92. New York et al.: ACM Press 1992, S. 63-76

[LaNa95] Lange, D.B.; Nakamura, Y.: Interactive Visualization of Design Patterns Can Help in Framework Understanding. In: Proceedings of the OOPSLA'95. New York: ACM 1995, S. 342-357

[Mey97] Meyer, B.: Object-Oriented Software Construction. 2nd. ed., Upper Saddle River, NJ: Prentice Hall 1997

[MCa86] McCarthy, J.: Applications of Circumscription to Formalizing Common-Sense Knowledge. In: Artificial Intelligence 28, 1986, pp. 89-116

[MDD80] McDermott, D.; Doyle, J.: Non-Monotonic Logic 1. In: Artificial Intelligence 13, 1980, pp. 41-72

[Ope94] The OpenDoc Design Team, OpenDoc Technical Summary. Component Integration Laboratories, 1994 (http://www.cilabs.org)

[Pla97] Platinum: Object Analysis and Design Facility Response to OMG/OA&D RFP-1. (http://www.omg.org/library/schedule/AD_RFP1.html)

[Rat97a] UML Semantics Appendix M2 - UML Meta-Metamodel. Vers. 1.0, 13 January 1997 (http://www.rational.com)

[Rat97b] Rational: Appendix M3: UML Meta-Metamodel Alignment with MOF and CDIF. Vers. 1.0, 13 January 1997 (http://www.rational.com)

[Rat97c] Rational: UML-Summary. Version 1.1. 09/01/1997 (http://www.rational.com)

[Rat97d] Rational: OCL. Version 1.1. 09/01/1997 (http://www.rational.com)

[Rat97e] Rational: UML-Semantics. Version 1.1. 09/01/1997 (http://www.rational.com)

[Rat97f] Rational: UML-Notation Guide. Version 01.09/01/1997 (http://www.rational.com)

[Rat97g] Rational: UML-Notation Guide. Version 1.0. 01/13/1997 (http://www.rational.com)

[Rat97h] Rational, Microsoft, Hewlett-Packard et al.: Object Constraint Language Specification (http://www.software.ibm.com/ad/ocl)

[Rum91] Rumbaugh, J. et al.: Object-oriented Modelling and Design. Englewood Cliffs, N.J.: Prentice Hall 1991

[SzOm93] Szypersky, C.; Omohundro, S.; Murer, S.: Engineering a Programming Language:

The Type and Class System of Sather. International Computer Science Institute Tech Rep. Tr-93-064, Berkeley, Ca. 1993

[WaNe95] Waldén, K.; Nerson, J.-M.: Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems. Hemel Hempstead: Prentice Hall 1995