UNIVERSITÄT
KOBLENZ · LANDAU

Institut für
Wirtschaftsinformatik

Fachbereich Informatik
Universität Koblenz-Landau

ULRICH FRANK

APPLYING THE MEMO-OML: GUIDELINES AND EXAMPLES

Juli 1998

Arbeitsberichte des Instituts für Wirtschaftsinformatik Nr. 11

UNIVERSITÄT
KOBLENZ · LANDAU

Institut für
Wirtschaftsinformatik

Fachbereich Informatik
Universität Koblenz-Landau

ULRICH FRANK

APPLYING THE MEMO-OML: GUIDELINES AND EXAMPLES

Juli 1998

Arbeitsberichte des Instituts für Wirtschaftsinformatik Nr. 11

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.

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

Bezugsquelle / Source of Supply:

Institut für Wirtschaftsinformatik
Universität Koblenz-Landau
Rheinau 1
56075 Koblenz

Tel.: 0261-287-2520
Fax: 0261-287-2521
Email: iwi@uni-koblenz.de
WWW: http://www.uni-koblenz.de/~iwi

Abstract

While a carefully designed modelling language should facilitate the development of useful object models, it is certainly not sufficient: The language does not tell much about how to find proper abstractions and how to decide between design alternatives. There are only a few general principles that help with the design of an object model. Design patterns that outline strategies for good design to meet certain requirements are certainly useful. However, they usually stress a high level of abstraction. In other words: They are only of limited help when it comes to analyse and model a particular domain. In the end, the development of object models remains a remarkable intellectual challenge. A great deal of the corresponding expertise can only be required by developing models - and by studying examples. This report presents a number of example object models which render domains typical for corporate information systems. The examples are designed using MEMO-OML (MEMO Object Modelling Language). Therefore, they also serve to illustrate the use of MEMO-OML which contains a few concepts that are not provided by other object-oriented modelling languages.

1. Introduction

MEMO-OML is a graphical language for the design of object models. Although its syntax and semantics are specified in a comprehensive metamodel [Fra98b] which is based on concepts of a meta-metamodel [Fra98a], its proper use is not a trivial task: Not only that some of the language concepts are difficult to understand, furthermore, representing a given or future domain by an appropriate object model can be a tremendous intellectual challenge. At the same time there are good reasons why we cannot expect a powerful theory that would tell us how to design good models. Against this background, this report serves to illustrate the use of MEMO-OML. For this purpose it will provide a number of small examples and three case studies. While we intend to demonstrate good modelling practice, there is no way to proof that the examples given below illustrate the best option. Therefore the reader should thoroughly reflect upon the arguments given for the recommended design choices. In order to design "good" models, we need an idea of quality. Therefore we will first look at the delicate problems related to model quality.

2. Some Remarks on the Quality of Object Models

It is a matter of experience the design of object models is a delicate matter. Firstly, there are problems with comparing different models and with describing their relationship to reality. Usually, different modellers will produce different object models of the same domain. At the same time, it is sometimes difficult to tell whether two different representations are equivalent. Also, it is not trivial to find out whether two models represent the same system (see [Bun74], pp. 101). Secondly, there are problems with judging the quality of a representation. This is due both to conflicting goals and the subjective nature of the matter. A conceptual model should serve as a medium to foster communication between the various participants in the process of system analysis and design. At the same time it should provide a solid foundation for implementing software. While the latter recommends - among other things - a high degree of formal rigour, making a model a medium for communication requires to take into account the conceptualizations the corresponding participants are familiar with. However, not only that it will be difficult to identify those conceptualizations, they will also vary from person to person. Additionally w have to take into account that every user of a model may learn over time: A model we do not understand at first sight may appear clear and even intuitive after we have studied it.

There is a growing awareness of the problems related to the evaluation of information models. Most of the corresponding publications ([KrLi95], [Lin94], [MoSh94], [Wie95]) have in common that they consider the evaluation of models to be a multi faceted problem. It requires to take into account the different conceptual and professional preferences of those who use a model. It should also take into account the represented domain and how it might change over. Against this background, it is evident that the examples presented above cannot satisfy every requirement that may be related to object models. Instead, they emphasize a “natural” representation that corresponds to common conceptualizations. Of course, that does not mean that everybody has to agree with our point of view. To illustrate our idea of a “natural” object-oriented representation, we will introduce a few basic guidelines.

3. Some Design Guidelines and Small Examples

Among the proponents of object-oriented software development it seems to be a self evident

fact that object-oriented concepts allow for a natural representation of reality: „People regard their environment in terms of objects. Therefore it is simple to think in the same way when it comes to designing a model." ([JaCh92], p. 42) There is evidence that this assumption is not appropriate in any case: Obviously there are people who feel more comfortable with traditional ER models or who find "rich pictures" (Checkland) more intuitive. From a software engineering point of view, it is often stressed that object-orientation fosters reusability – mainly by providing inheritance and encapsulation. While we sympathize with both opinions, we do not think that they focus directly on the essential advantage of object-oriented modelling. Instead, we consider another feature to offer the key advantage over traditional data/functional modelling: Object-oriented modelling allows for a substantial higher level of *semantic abstraction*. It is this feature that, in turn, contributes to more intuitive and reusable artefacts.

3.1 Semantic Abstraction as the Essential Advantage of Object-Oriented Modelling

Abstraction allows to neglect details that are regarded as irrelevant or that are subject of future change. We speak of semantic abstraction, if abstraction does not compromise information content. For instance: Concepts like "String" or "Integer" to describe aspects of real world domains can be regarded as an abstraction, too. However, since such concepts allow for multiple interpretations, their semantics is very little. Semantic abstraction allows to discriminate between concepts that are regarded as different. As most object-oriented modelling languages, MEMO-OML fosters semantic abstraction mainly by three concepts:

Information hiding

Information hiding is directly provided by the encapsulation of an objects internal structure. While it is a common - and useful - practice to differentiate between attributes and services as early as possible, it may happen that one does not know whether a particular feature will be stored in an attribute or computed by a services for the lifetime of a class. In this case, encapsulation allows to specify the external view on an object, its interface - and thereby to abstract from possible future changes. Fig. 1 shows a simple example: The retail price of a product may be calculated from the wholesale price - or it may be assigned explicitly. Sometimes the interface of a class includes services which might confuse a programmer who is going to reuse this class. In case the programming language used for the implementation of the class allows to express different degrees of visibility, those services could be made invisible for external use. MEMO-OML allows for three degrees of visibility (or accessability, "public", "protected", and "private". Fig. 2 shows an example where services of a class are assigned different degrees of visibility. While "public" is to express that the corresponding service is visible to anybody, "protected" restricts the access to objects with certain privileges, "private" means that the service can be accessed internally only. Notice, however, this is only the suggested meaning. MEMO-OML does not include a specification: Usually, this feature will become relevant only after implementation. Hence, the semantics will depend on the implementation language anyway - provided the implementation language offers these concepts at all.

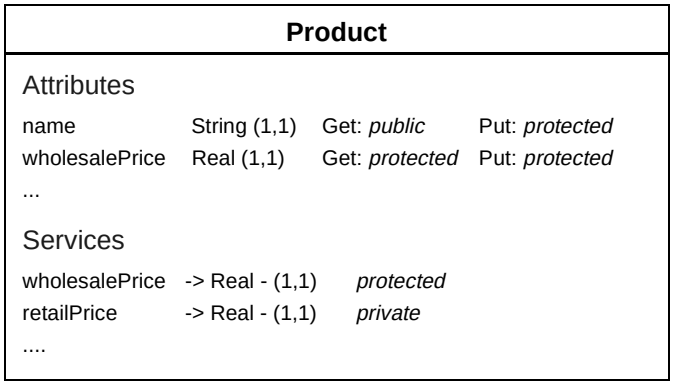

Fig. 1: Information hiding through encapsulation

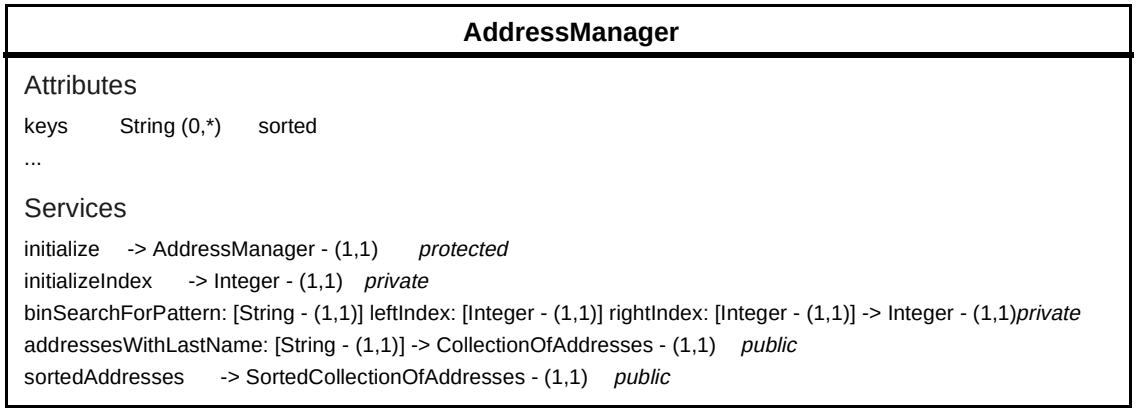

Fig. 2: Information hiding by private and protected services

Delaying (implementation) decisions

Analysing the terminologie within a domain of discourse will often lead to a generalisation hierarchy. Sometimes generalising over a set of classes leads to generic terms that denote in fact abstract classes: They are important to make propositions (or assign properties) that apply to all subclasses. However, they are abstractions in a sense that there is no instance that would not belong to a more specific class. Sometimes it is a good idea to introduce features of an abstract class even if they cannot be specified on such a generic level: In case those features are essential for the generic term, they inform about relevant properties any subclass has to offer - in a way that may be specified within the subclass. Using abstract classes and deferred features means not to neglect generic/essential features of an abstract class. Instead, it allows to be as specific as possible about a generic concept. Thereby it is possible to delay decisions that have to be made later on within the subclasses.

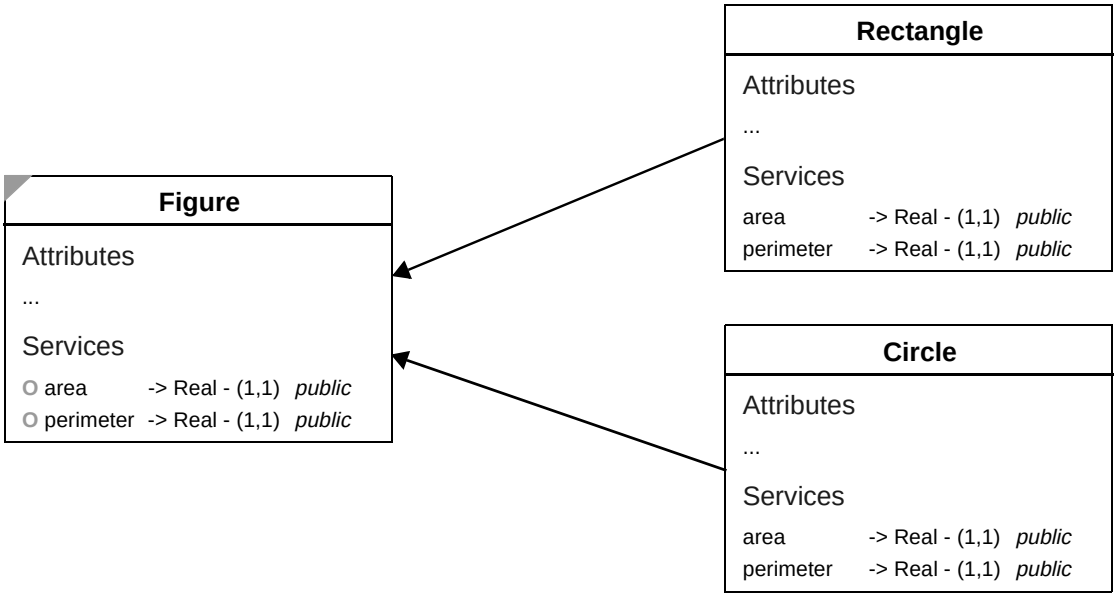

Fig. 3: Abstraction through Abstract Classes and Deferred Services

Defining domain level concepts

Different from traditional entity relationship modelling, object-oriented modelling languages often allow to specify attributes not only by a small number of base level types (like String, Integer, Real etc.), but also by classes that have been specified by the modeller. Introducing domain level classes to specify attributes comes with two major advantages. Firstly, it allows for a higher level of semantic abstraction which contributes to a higher level of integrity - by eliminating interpretations which are regarded as inappropriate or harmful (like in fig. 4, left example). Secondly, if fosters a higher degree of flexibility because the specification of the domain level classes may be changed in time without effecting the specification of the class the corresponding attribute is part of (see fig. 5).

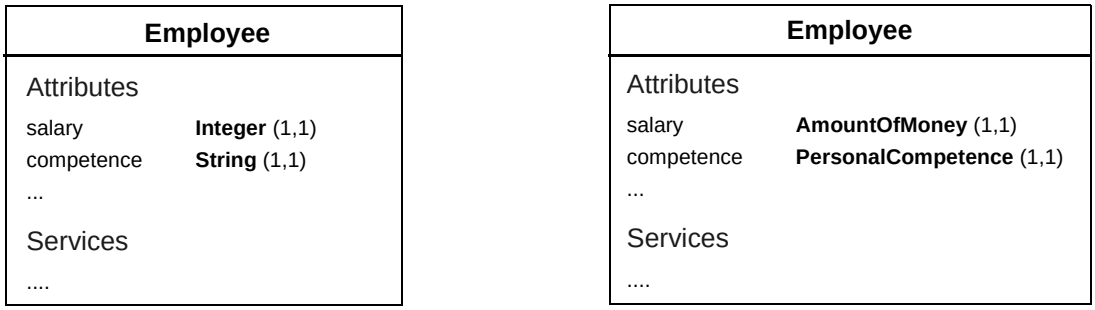

Fig. 4: Increasing the Level of Semantic Abstraction by Domain Level Classes (right example)

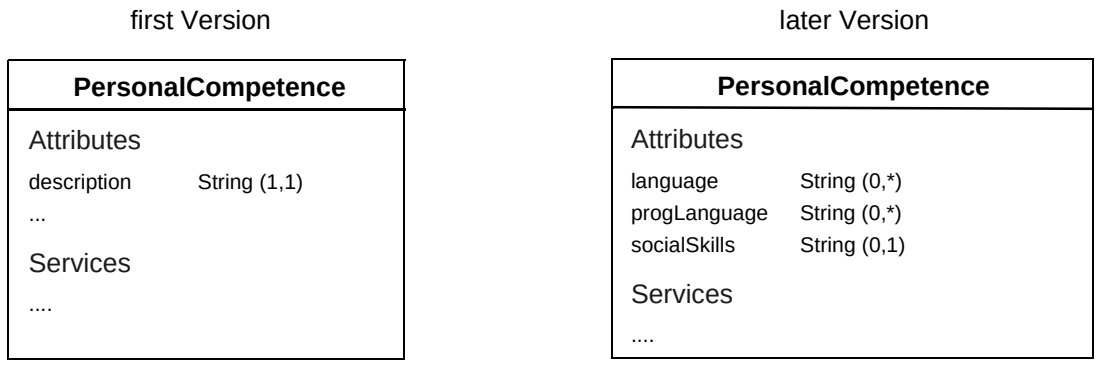

Fig. 5: Fostering Flexibility/Maintenance by Semantic Abstraction

3.2 Some Basic Guidelines

Numerous decisions have to be made during object-oriented analysis and design. While some of them are rather specific (and subtle) others are of a more general nature.

Attributes vs. Associated Objects

One of the more general problems concerns the question whether or not a particular aspect ought to be represented by an attribute or by an associated object. One could argue that this is not a problem at all because in the end - on the implementation level - there is often (depending on the implementation language) no difference between attributes and associated objects anyway. However, independent from the concepts offered by a programming language, there is a clear difference between attributes and associated objects on the conceptual level which can be expressed in the following rule:

> An attribute is something that does not have an identity of its own independent from the object it belongs to. An associated object, on the other hand, has an identity of its own independent from the object it might belong to at first sight.

Such a distinction is of crucial importance for the proper design of software: While the object that represents an attribute exists only within the object it is part of, an associated object may be referenced by other objects as well. Therefore, the inappropriate use of attributes contributes to redundancy, while the inappropriate use of associated objects can be a thread to an object's integrity. In most cases, the rule expressed above can be applied in a straightforward way. Sometimes, however, the decision whether to use an attribute or an associated object depends on assumptions about the domain and on the evaluation of the problems that might be caused by redundancy. Fig. 6 shows a number of examples (*lastname*, *salary*, *department* ...) where the decision between attributes and associated objects is obvious (although not imperative). This is different with *address*. While there is no doubt that an address exists without a person who lives at this particular location, it may be regarded as sufficient to model it as an attribute: It may hardly happen that there are to persons with the same address represented in the system to be designed. Also, the implementation of an associated object is usually more expensive (semantics of delete operation, bi-directional associations).

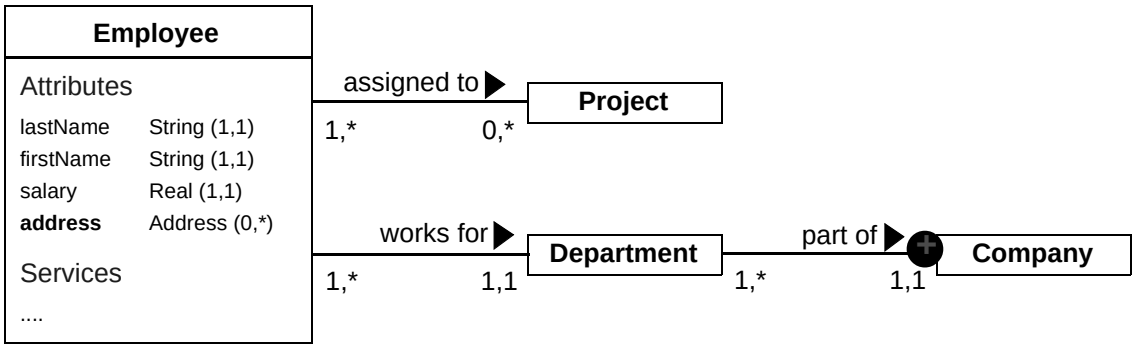

Fig. 6: Differentiating between Attributes and Associated Objects

Interaction vs. Aggregation

After the decision for an association has been made, sometimes the question will occur whether to use an interaction or an aggregation. From a common sense point of view, the distinction between these two options seems to be trivial: Whenever an association can be labeled with a designator like "is part of", "consists of", "belongs to" etc., it is a candidate for an aggregation. However, there is hardly a difference between the semantics of both concepts, as they are specified within MEMO-OML. In the end, there is only one necessary feature and one modest constraint that must apply for any aggregation ([Fra98b], p. 16):

#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.

Notice, that in a particular case the constraint may also apply to an interaction. Therefore, we recommend the following rule of thumb:

> An aggregation should be used only, if the common sense notion of aggregation or containment applies *and* the formal constraint #2 is valid.

Inheritance vs. Delegation

Without any doubt, inheritance is an outstanding feature of object-oriented design. Not only that generalization and specialization foster maintainability and reusability. Furthermore, generalization can be regarded as a common sense concept, thereby fostering an intuitive and natural way to describe the real world. However, in some cases inheritance, although applied in an intuitive way, can result in inappropriate concepts. Consider the following example: In order to design an information system for a university, you need objects to represent students, research assistants, professors, etc. Since they share common features like name, date of birth, sex, etc., you would introduce person as a generalization - resulting in rather natural concepts: a student *is a* person, a professor *is a* person, etc. Then you find out that you need objects to represent programmers, lecturers, administrators, etc. Again inheritance seems to be the right choice: Apparently programmers, lecturers, and administrators happen to be persons.

However, students as well as research assistants or professors may also be programmers - or even programmers and lecturers at the same time. While, for obvious reasons, single inheritance is not an option in this case, multiple inheritance would allow to express those semantic relationships (see fig. 7).

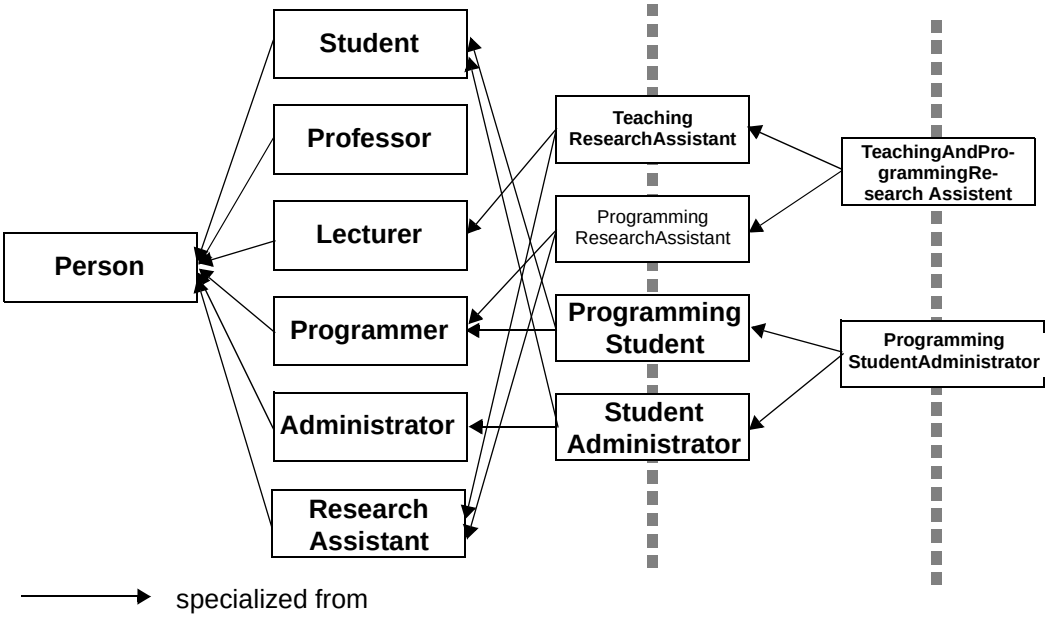

Fig. 7: Concepts resulting from Multiple Inheritance

The classes defined in this hierarchy would in principle allow to express the combinations of responsibilities mentioned above. Unfortunately, it results in concepts you would hardly consider as a natural way of modeling the world - like "teaching and programming research assistant". However, even more important is the fact that inheritance - no matter whether it is single or multiple - will lead to misconceptions that jeopardize a system's maintainability and integrity. Think of a person that may be regarded as a programmer in one context, as a student in another context. With most object-oriented programming languages, inheritance is specified in a way that, in our case, would result in instantiating objects from different classes. Hence, the same person would be represented by different objects. In our opinion, this sort of redundancy is not acceptable.

As this small example illustrates, using inheritance may result in inadequate models, although every single "is a"-relationship seems to be appropriate. This rather confusing phenomenon is caused both by the ambiguity of "is a" in natural language and the implementation of inheritance in common object-oriented programming languages. Natural language often does not explicitly differentiate between a concept and its instances. This is different with programming languages. In most languages we know, "is a" is related to a set of features a class shares with its subclasses. An instance, however, usually is of one and only one class. In other words: Within object-oriented programming languages, an instance of a the class is (usually) not an instance of the respective superclass.

Beside redundancy, *lack of flexibility* is another shortcoming of inheritance. When we talk about a domain like the one outlined above, we obviously use abstractions that depend on the

current *context* we are in. Sometimes we are interested in a person being a lecturer, and we do not care whether he is able to write a program or not. In another context we may regard the same person as a system administrator. Inheritance, however, does not allow to express changing contexts that may apply during the lifetime of objects. In other words: Generalization requires to "freeze" certain abstractions before having instantiated a single object, while we sometimes need concepts that allow to change abstractions after objects have been instantiated.

It is one of the characteristic features of MEMO-OML that it offers an alternative to inheritance. 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"). 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 (for a comprehensive specification see [Fra98b], pp. 23, pp. 45). The following examples illustrate the use of delegation.

a) Managing lectures at a university

Suppose there is a set of lectures defined within the curriculum. A lecture is characterized by a title, a table of content, an abstract, and maybe associations to other lectures. By default a certain lecture (like "Introduction to Operations Research") is offered by exactly one professor. Furthermore, we have to deal with concrete lectures offered in a particular semester. While these concrete lectures are characterized by the same properties as the corresponding "essential" lectures mentioned before (note that natural language hardly allows to avoid ambiguity here), they have to be assigned additional information: time, location, maybe students ... By modelling concrete lectures as roles of essential lectures we would accomplish exactly what is required in this case. If it may happen that the professor assigned by default can be substituted with somebody else for a concrete lecture, we need to add an association between a concrete lecture and a professor.

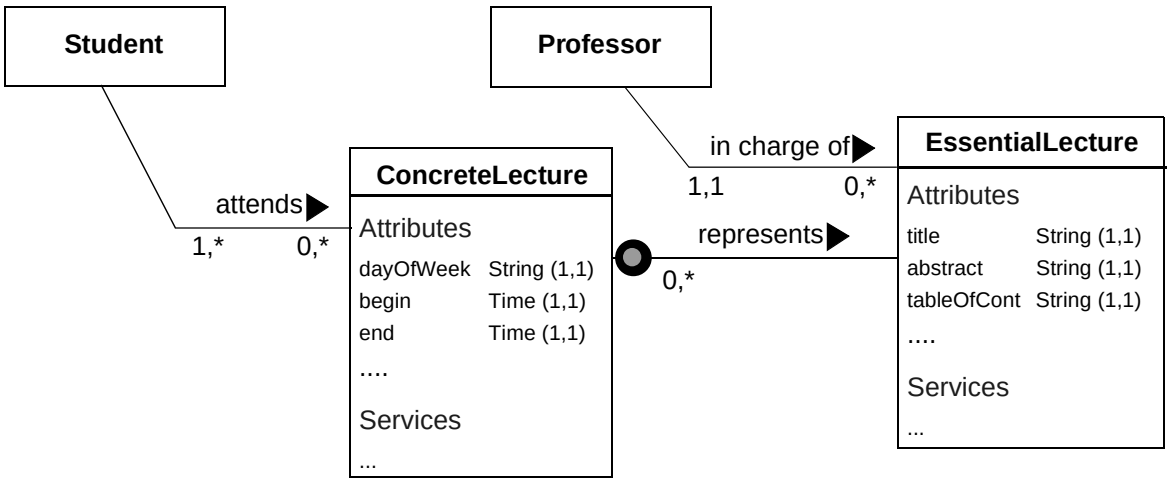

Fig. 8: Essential and concrete lectures

b) "Class Migration"

An insurance company wants to keep track of future customers by storing information about its current customers' children. Once the children turn 18, they are to be offered insurance services specially designed for young people. If they eventually become customers, there is need to update the company's database. In a straightforward approach, one would probably delete the particular instance of the class Dependant and instantiate a new instance of InsuredPerson. Afterwards you would have to initialize this instance using the relevant parts of the former Dependant instance. However, not only that this approach is somewhat cumbersome, it also jeopardizes system integrity (there may be numerous references pointing to the Dependant instance). A more ambitious approach would aim at changing an object's class - from Dependant to InsuredPerson in our case. Such an approach, usually referred to as "Class Migration" (see for instance [Wier95]), is rather confusing (what does it mean when something "changes" the concept it is defined by?). Furthermore, it will usually be a remarkable effort to provide for a satisfactory implementation. This is different with delegation. We could regard both an instance of InsuredPerson and an instance of Dependant as roles of an instance of Person (see fig. 9). In this case, we would simply add a new role by creating an instance of InsuredPerson. Since the multiplicity for Role is 0,1 in our example, the instance of Dependant would now have to be deleted. This would, however, not affect relationships between customers as long as those are modeled as associations between Person objects.

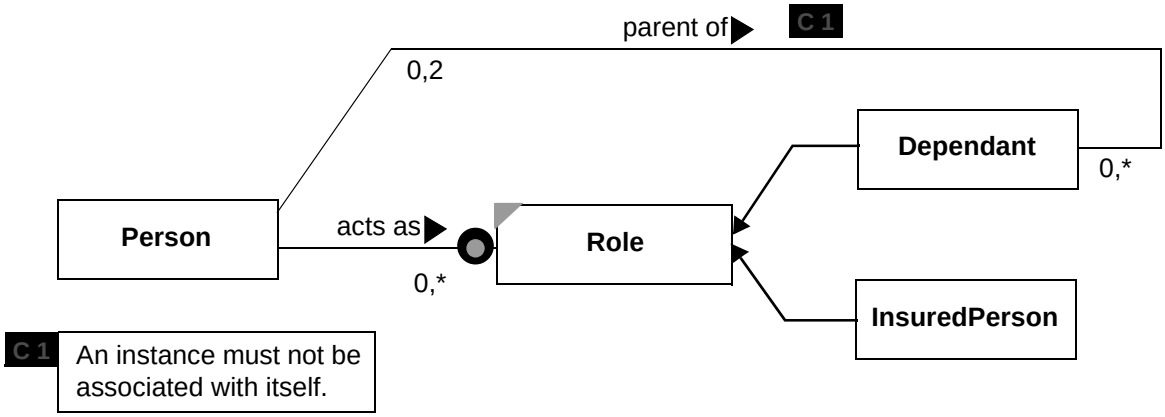

Fig. 9: Avoiding Class Migration through Delegation

c) "Multiple" role filler classes

A retail company serves both individuals and companies. Some of those companies act as suppliers as well. If we first look at the second aspect, it would be a good idea to regard a customer as a role of a company. Supplier could then be another role a company may play. However, an individual may be a customer as well. Treating both a company and a person as role filler of the role customer is not permitted without further consideration: It would not be compliant with constraint that, at a point in time, a role object must not be associated with more than one role filler object (see [Fra98b], p. 25). On the other hand, it may turn out that introducing two different kinds of customers without a common superclass will add redundancy, since there may be numerous aspects of customers that do not require to check whether they are individuals or companies. In order to take advantage of the benefits offered by delegation, there is only one chance left: introducing a common superclass of the role filler classes Person and Company. This class may be an abstract class, for instance AbstractPerson. It should offer essential features of both Person and Company - such as name and address. No matter whether a particular instance of Customer is associated with a Company or a Person object, it would be able to answer to the protocol defined in AbstractPerson. Note that the maximum multiplicity of the role filler class, AbstractPerson, prevents a Customer object being associated with a Company object and a Person object at the same time. However, this example should illustrate that delegation is not always the best choice. Only if it is acceptable to introduce a common superclass of role filler class candidates (that means if there is at least a few common features), delegation is an option.

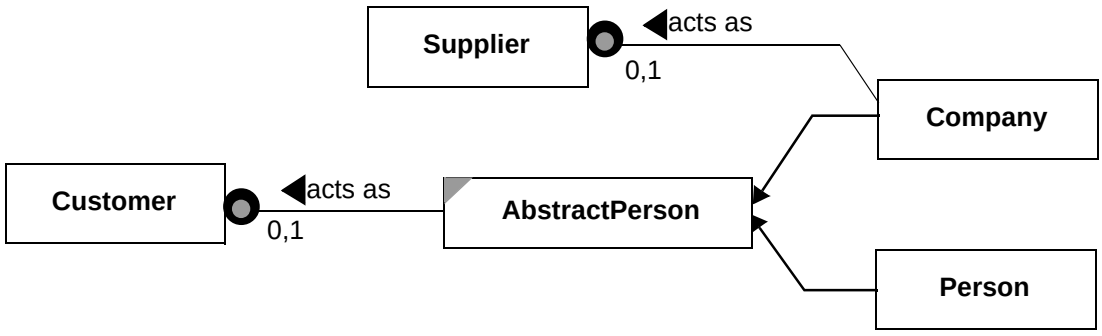

Fig. 10: "Multiple" Delegation through Generalisation

d) Multi level delegation vs. inheritance

The dean of a university faculty has to be a professor and an employee as well. At first sight, it seems to be satisfactory to model dean, professor, and employee as roles of a person. However, such a model would lack relevant semantics, since it would not tell that somebody can only become a dean if he is a professor. Specializing dean from professor via inheritance is usually no convincing option: With his role as a professor somebody may have a different room, secretary, and phone number than with his role as a dean. For this reason, we would need two instances - one representing a professor, the other representing a dean. In order to express the fact that a dean has to be a professor, dean would be modelled as a role of professor. What about the relationship between professor and employee? Whether one should use inheritance here or delegation again (which would result in "multi level delegation") can hardly be answered in a general way. Modelling professor as an employee's role would certainly be more versatile: You could delete the role object without deleting the corresponding Employee object. However, if you wanted to stress that a professor will be a professor as long and only as long as he is an employee, inheritance would be the preferable option: After deleting a Professor object its employee features would be deleted as well. On the other hand Professor would inherit all other possible roles of Employee. In the end the option to decide for depends on your notion of a professor. In case you consider a professorship a lifetime academic position, regardless of a corresponding occupation, delegation would definitely be a better choice. However, then you would have to model professor as a role of a person, not of an employee - thereby loosing the information that by all means a dean has to be an employee. It might be an acceptable compromise to differentiate between an employed professor, an emeritus, and maybe a visiting professor (see fig. 11). Note, however, that this compromise would lack a common abstraction of the three types of professors: If you made Professor a subclass of AbstractProfessor, it would inherit to be a role of Person - thereby excluding that it could be a role of Employee (multiple delegation is not permitted by definition, see [Fra98b], p. 25). At the same time Professor could not be a subclass of Employee - as long as you do not allow for multiple inheritance.

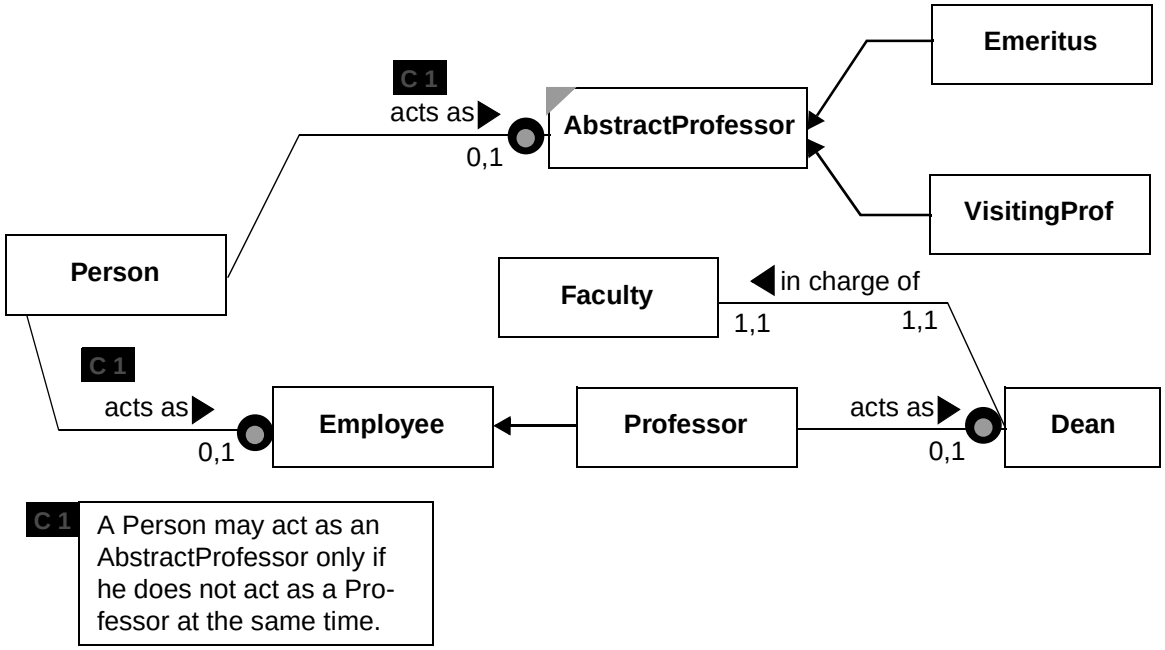

Fig. 11: Combining multi level delegation and inheritance

Although delegation is often more appropriate than inheritance, it does not replace inheritance. The following guidelines describe when delegation is a good option.

- Do not get confused by the ambiguity of "is a". Ask yourself whether a relationship between two concepts could also be called "represents" or "acts as" respectively. If this is the case, you have found a delegation candidate.
- Delegation is closely related to the common sense concept of a role. The existence of a role may be indicated by notions such as "task", "job", "serves as", "works as", etc. Therefore you should look for corresponding terms within available descriptions of a domain.
- A generalisation that does not necessarily hold for the entire life time of the system to be designed could be a case for delegation. For instance: If a professor does not have to be a an employee by all means, delegation will be a better choice than inheritance.
- Whenever you encounter the existence of different views on an object, or different contexts an object may be assigned to, it is a good idea to check whether these views or contexts can be related to roles or responsibilities of the object in a natural way. In this case, delegation might be a useful option.
- Some real world entities are likely candidates for becoming role filler objects: persons, organizations, and versatile machines. Assigning the objects of a preliminary object model to such categories may help with identifying delegation associations.

Whenever the common sense notion of generalisation/specialisation applies *and* delegation is not an option, inheritance will usually be a good choice. Nevertheless, the question remains how to apply inheritance. Consider the example in fig. 12. Is it preferable to specialize Square from Rectangle or to do it the other way around? Specialising Rectangle from Square allows

to refine the specialised class in a common way. One would add an attribute to represent the second side. Additionally, the services that compute the area and the perimeter would have to be refined. From a programmer's point of view, this may not only be an acceptable but the preferred way to apply inheritance. However, we advise strongly against this sort of inheritance, sometimes called "implementation inheritance". Instead, we recommend "conceptual inheritance" which means to use generalisation/specialisation in an intuitive way (that corresponds to our own conceptualisation). It is common sense that of both terms rectangle is the generic one. Therefore, Square should be specialised from Rectangle. Conceptual inheritance fosters the understandability and the maintainability of an object model. But how would one express that all sides of a square have the same length? Since MEMO-OML does not allow to remove an inherited attribute, we could redefine the services that change the value of both sides. It is, however, better to use a *guard* to express this constraint. A guard offeres a higher level of abstraction than services. Hence, it is sufficient to specify the constraint only once - and not again a again with every service that is concerned. Thereby a constraint also fosters the integrity of a class over time because the specification of additional services does not require to explicitly take the constraint into account. Fig. 12 shows a preliminary notation for the specification of the corresponding guard (in its current version, MEMO-OML does not include a formal specification language).

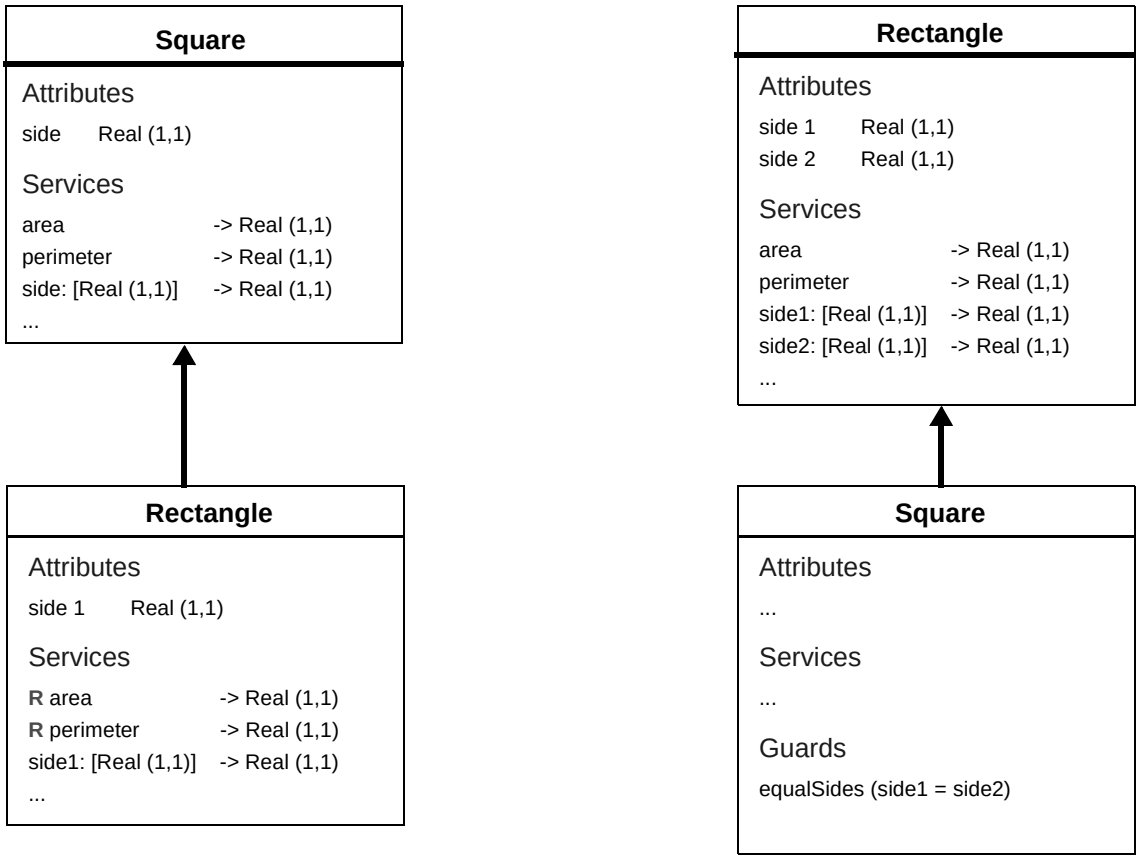

Fig. 12: Conceptual vs. Implementation Inheritance

3.3 Metaclasses and Typed Collections

Metaclass is a concept that is hardly used during analysis because it emphasizes a level of abstraction that very likely to be confusing at this stage. Nevertheless, it can be helpful to introduce metaclasses later during the design of an object model. Firstly, a metaclass allows to specify information that should be stored within its sole instance, a class, rather than within the objects of this class. This is typically information about the population of instances, like their number or other statistical quantities. Secondly, it is possible to use metaclasses to add flexibility to the specification of the corresponding class over time. The metaclass shown in fig. 13 includes alternative services to instantiate and initialize objects from its corresponding class.

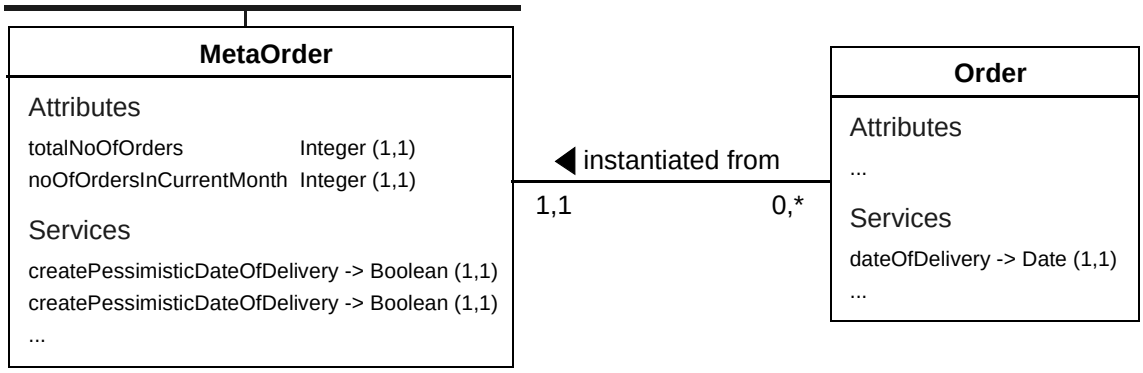

Fig. 13: Additional Information and Flexibility through Metaclasses

Collections of objects, such as bags, sets or dictionaries are usually no subject of analysis. Instead, the need for multiple instances will usually be expressed through multiplicities - for attributes, parameters, returned objects or associated objects. During late stages of design, there may be need for modelling collections. Those collections will usually be typed, i.e. their content is restricted to objects of certain classes. At the same time, it is desirable to reuse existing, more generic collection classes. Fig. 14 illustrates how to model a dictionary that contains instances of Integer as keys and instances of Person as values. Specialising DictionaryOfPerson from Dictionary requires to redefine the classes of the associated objects. The notation suggested by MEMO-OML requires to add a textual specification to a specialised class. The specification defines how the classes of objects associated with this specialised class are redefined (according to the covariance rule). Notice that the redefined services do not necessarily require a complete re-implementation. Instead, it may be sufficient to redefine the pre- and postconditions only. If a tool is used to build a model, the redefinitions of services resulting from redefining associated objects could be generated automatically.

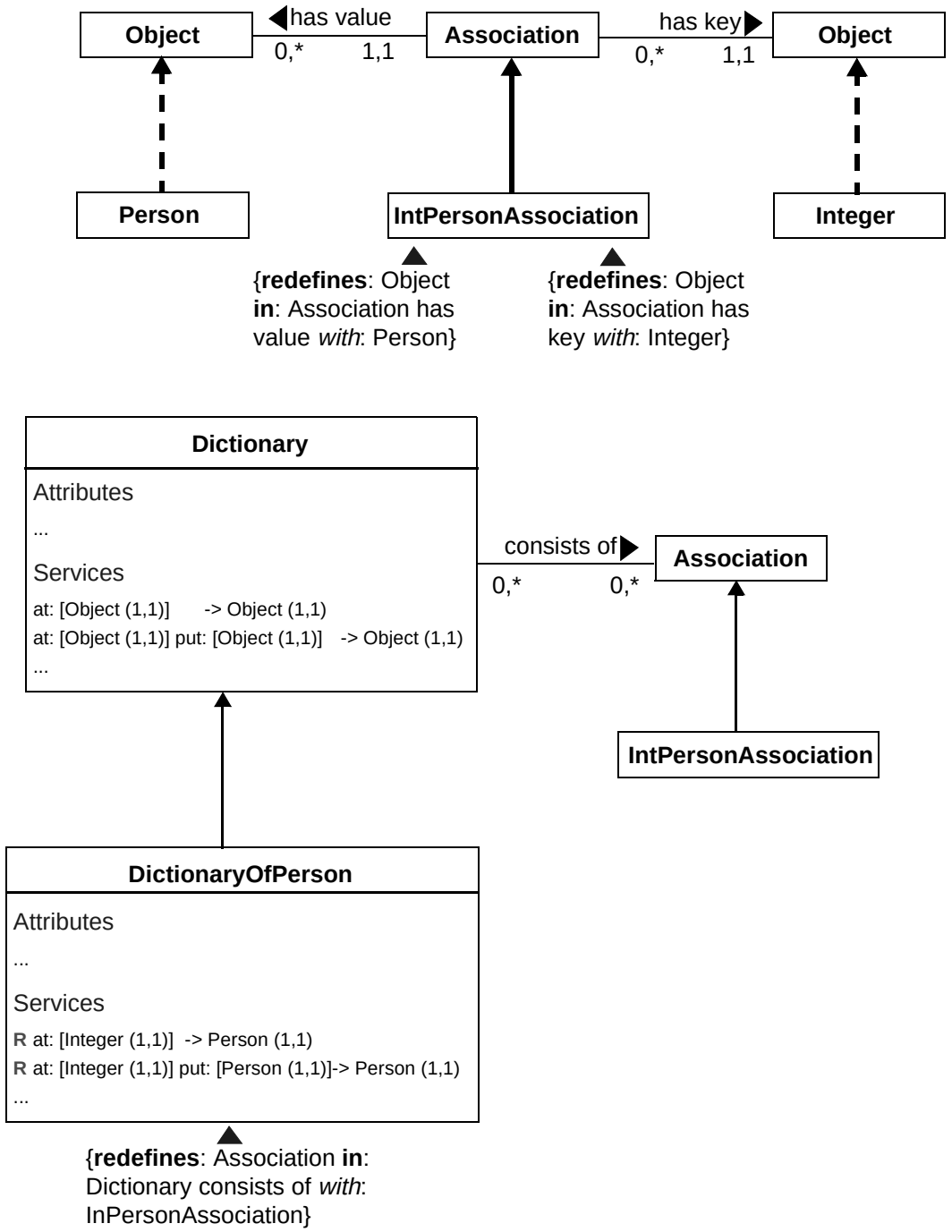

Fig. 14: Conceptual vs. Implementation Inheritance - only to illustrate

4. Case Studies

In order to illustrate the design of object models with MEMO-OML, we will consider a few small case studies. They focus on domains, MEMO is primarily intended to be used for: corporate information systems. While they reflect a certain style of modelling, they should not be

regarded as ideal models: Evaluating a model is a delicate task and depends - among other things - on the context they are used in as well as on individual preferences and arbitrary decisions which can hardly be justified in a rational way.

4.1 Addresses

The following model aims at a subject that is part of most corporate information systems: addresses and communication media of persons, employees, organisations or organisational units respectively. It should also fit the requirements imposed by international addresses.

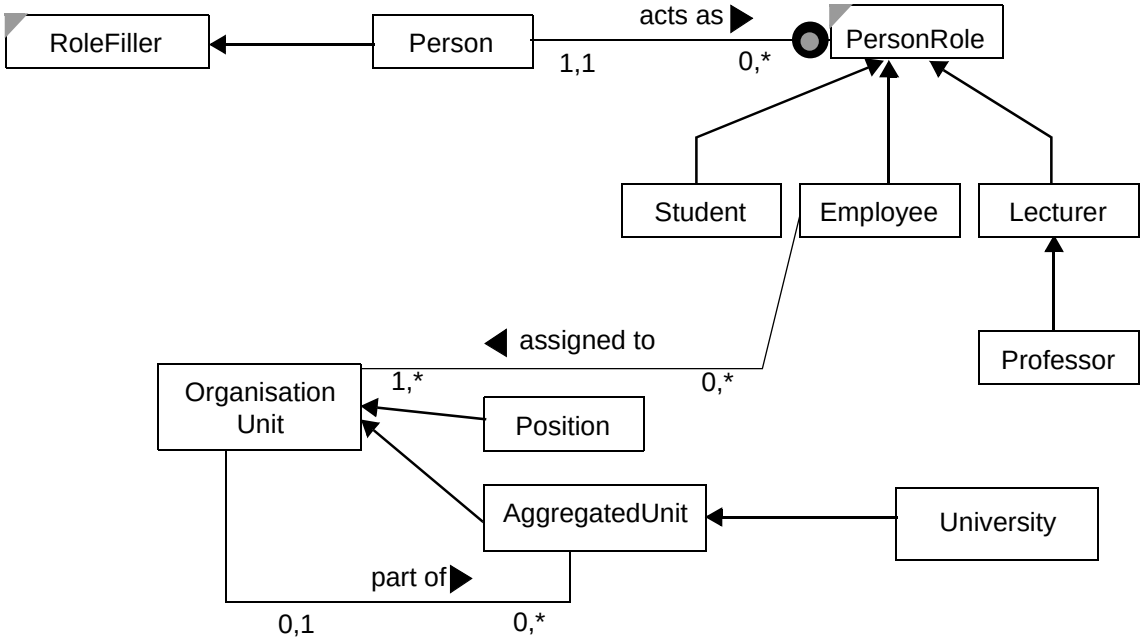

Fig. 15: Objects assigned to an Address

Before we specify the classes rendered in fig. 15 in more detail, we will conceptualize the notion of an address. While we can assume that the specific structure of an address may vary from country to country, certain features are always required. For instance: An address contains the name of the city or country it belongs to. While one could model both country and city simply as attributes - each of which specified as String - we rather use a higher level of abstraction that will allow for more flexibility and less redundancy. For this purpose we introduce special classes (Country and City). Instead of using them to specify attributes we use them to define associated objects, since both cities and countries can be expected to have an identity independent from a particular address. In order to allow for various types of addresses, the essential features of an address are modelled using a common abstraction (GenericAddress) which can be specialised into country specific types of addresses. A particular country should be associated only with an address instantiated from a class that had been introduced for this country. Only if no such class exists, it may also be associated with an address of another class, for instance one of GenericClass. Notice that a respective constraint can hardly be specified: The

country a specialised address class is meant for, can only be identified by the name of the class while the corresponding country would have to be identified by the state of an instance attribute (i.e. "name"). The name of an address class, however, can hardly be used for a general constraint. Enumerating all class and country names, on the other hand, would be rather cumbersome. For this reason the constraint is not specified but only added as a comment. In case, one does not know all relevant countries in advance, it would increase the system's flexibility to allow for the specification of new classes. While this could be expressed in MEMO-OML using the metaclass concept, we presume that all the countries required within the system to be developed are known at the time of designing the object model.

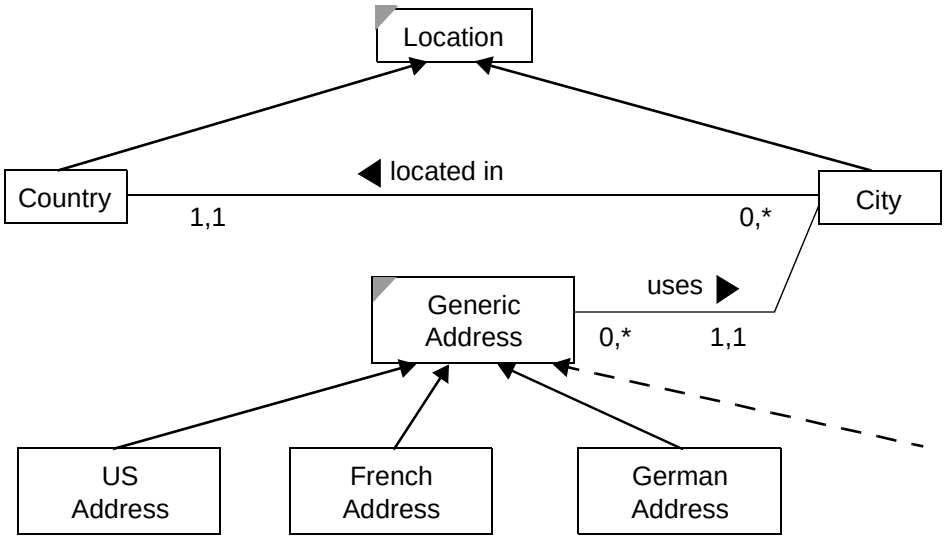

Fig. 16: National Addresses

Communication media, such as telephone, fax, or e-mail, are usually regarded as important parts of addresses. However, with the increasing relevance of mobile communication devices it would not be a good idea to model communication media as part of addresses. Furthermore, even stationary devices cannot always be uniquely assigned to an address. For instance: Two people may share the same address, but they may have different numbers for stationary phones at the same time. Before we consider how to associate communication media with addresses or persons, we will first focus on objects only that serve to represent communication media.

Telecommunication is the superclass of both mobile communication (MobileCom) and non mobile communication (NonMobileCom). Notice that those classes do not represent communication devices like telephone or fax directly. Instead they represent communication media which can be reached through an address (telecommunication number). In order to allow for the differentiation of devices, Telephone and Fax are added as roles of Telecommunication. Both can be connected to a mobile communication medium or to a non mobile medium. In order to reduce redundancy (dialling code) - and to allow for additional information - MobileCom is associated with a network provider (ComNetwork). The dialling code required for non mobile communication is stored with instances of City and Country, which are associated with instances of NonMobileCom via CombinedAddress and GenericAddress respectively. ServiceCom has been introduced as an additional medium that is part of a customer service. In this

case, it does not matter whether the medium is mobile or not. The essential feature is the service - for instance: toll-free call - that is provided for the caller. There are two specialisations of NonMobileCom, AutonomousCom and Extension. They serve to express that numbers within organisations are often constructed from a base number (for the organisational unit) and an extension.

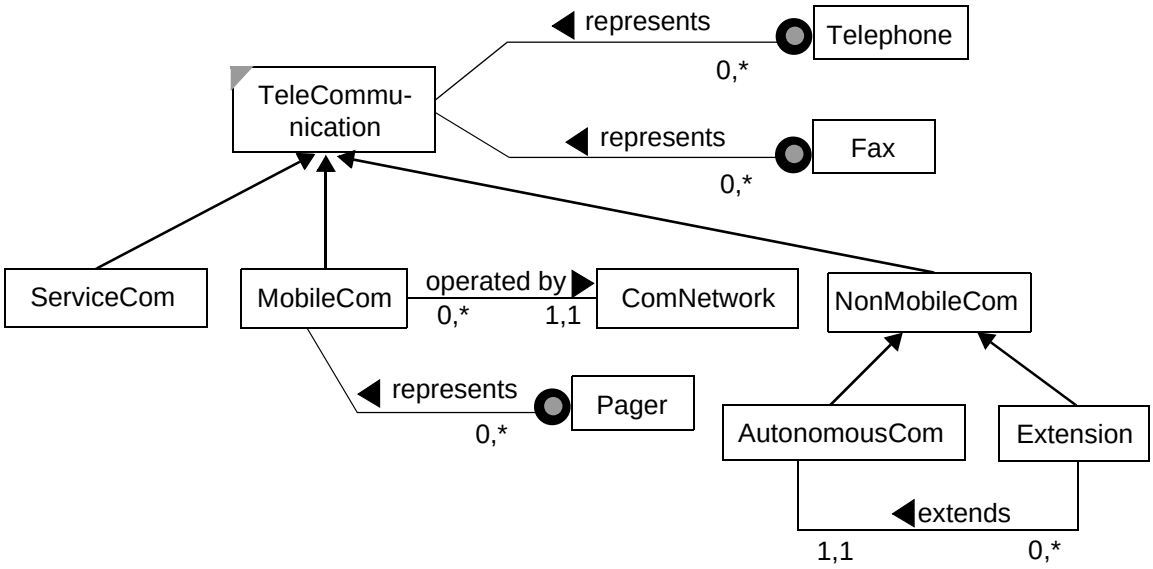

Fig. 17: Communication Media

Assigning communication media to people working in an organisation can happen on different levels of abstraction. Firstly, an employee may have a number that is associated with his position (a special OrganisationalUnit). Secondly, he may have a non-mobile number that he would keep independent of his position (that would probably be the default in most cases). Finally, he may have a mobile communication device. In addition to that, every person (independent of the role he or she holds) can be assigned both, mobile and non-mobile communication devices. An organisational unit can be assigned a mobile communication device, too (see fig. 18). The part of the model that renders electronic communication media such as e-mail and web pages is of preliminary nature. While it would make sense to differentiate between the various parts of an address (like country, domain, etc.), there is so much diversity in real life (for instance: employees sometimes have e-mail addresses and web pages that are not associated with their employer) that we do not consider addresses in detail.

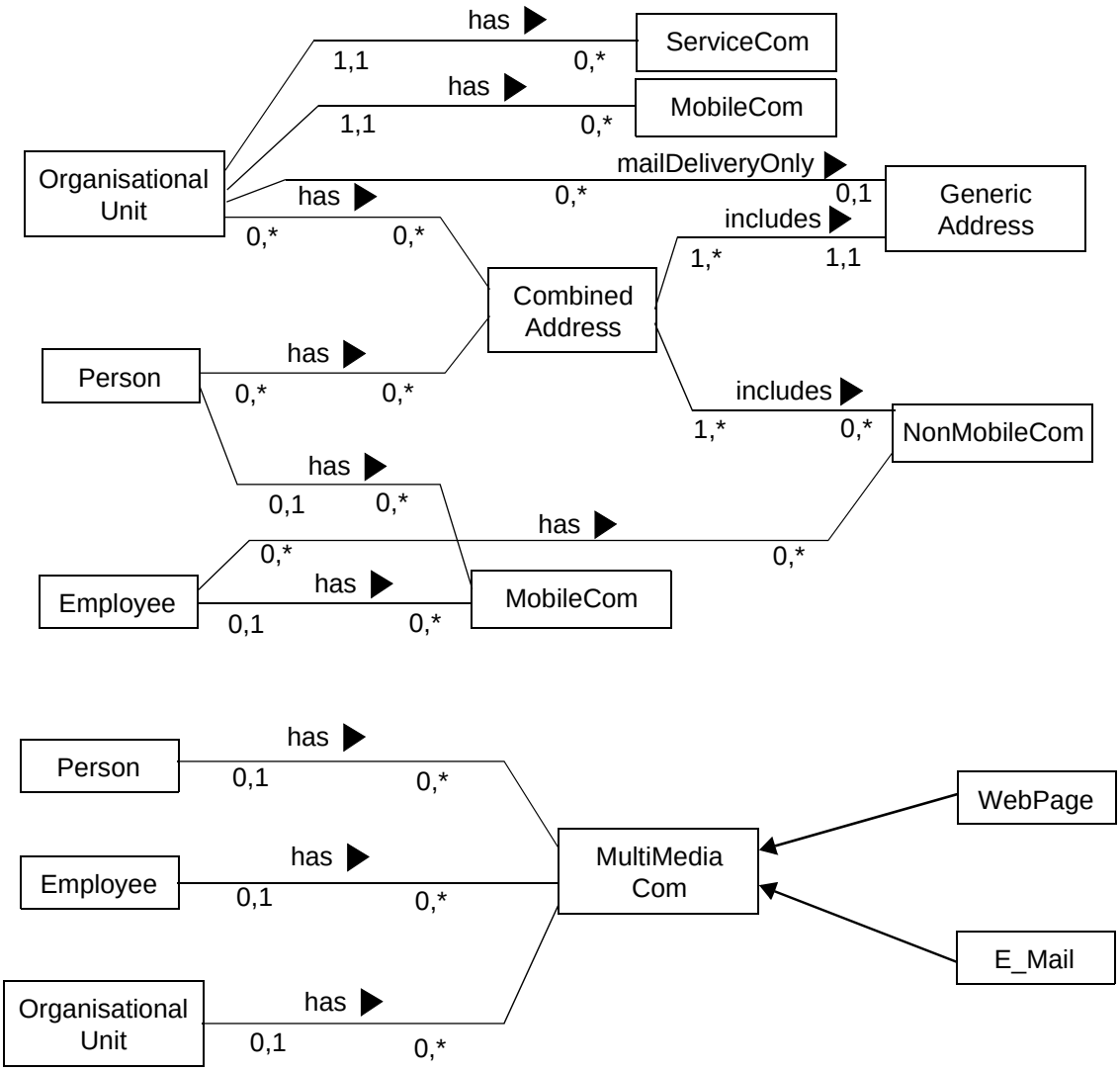

Fig. 18: Communication Media and their Owners

To give a better impression of the classes' semantics, we will specify a few selected classes in more detail (see fig. 19).

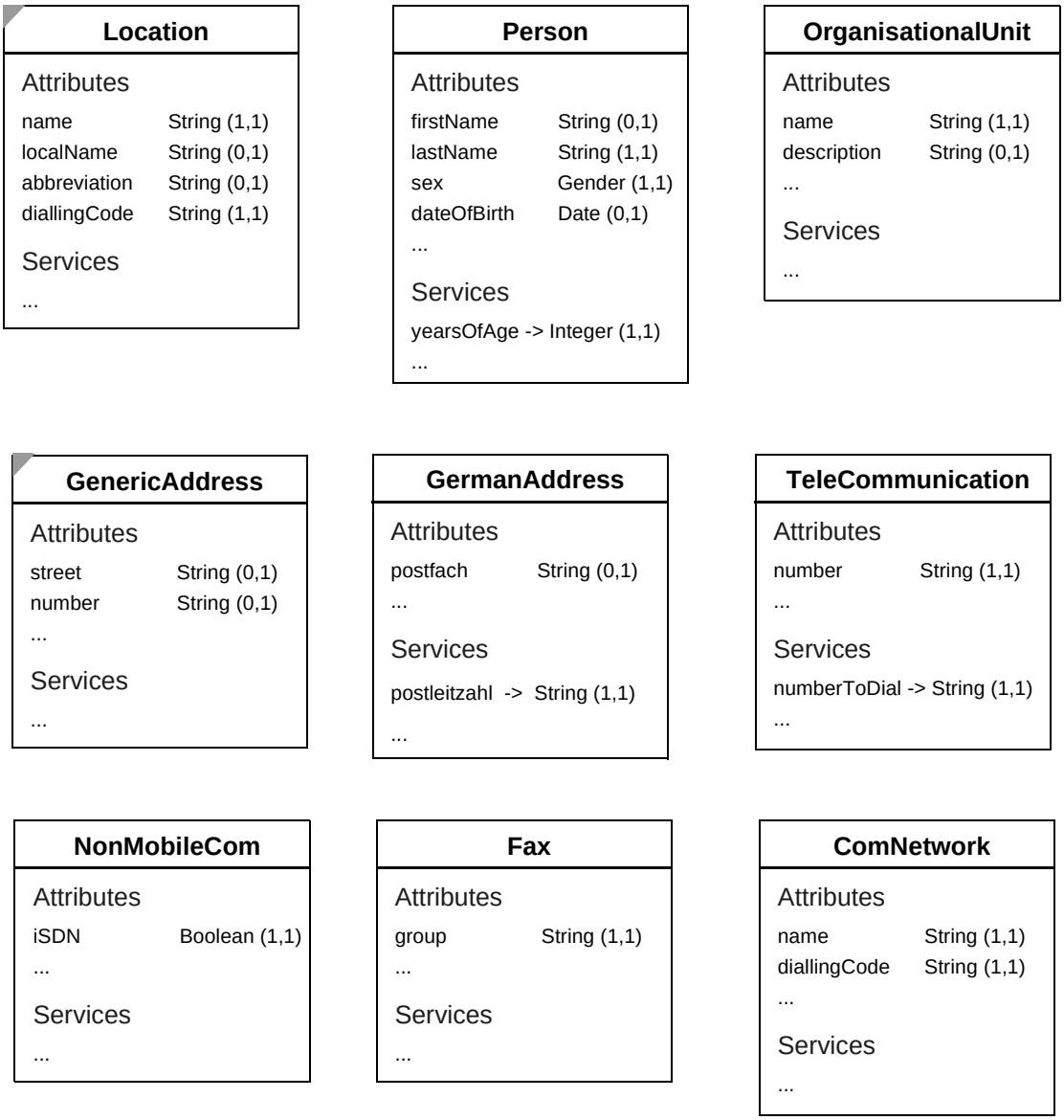

Fig. 19: Detailed Description of selected Classes

Notice that the specification of classes in the object model does neither include instance variables that are used to store references to associated objects, nor services that allow to access those associated objects. While instance variables that are only used to store references must not be modelled as attributes in order to avoid confusion (on a conceptual level there is an important difference between attributes and associated objects), access services could in principle be added. However, usually this information can implicitly be provided by the specification of an association anyway. This requires to specify whether or not the instances of a class that participates in an association should store a reference to associated objects. Fig. 20 renders a refinement of a part of the object model that includes these specifications.

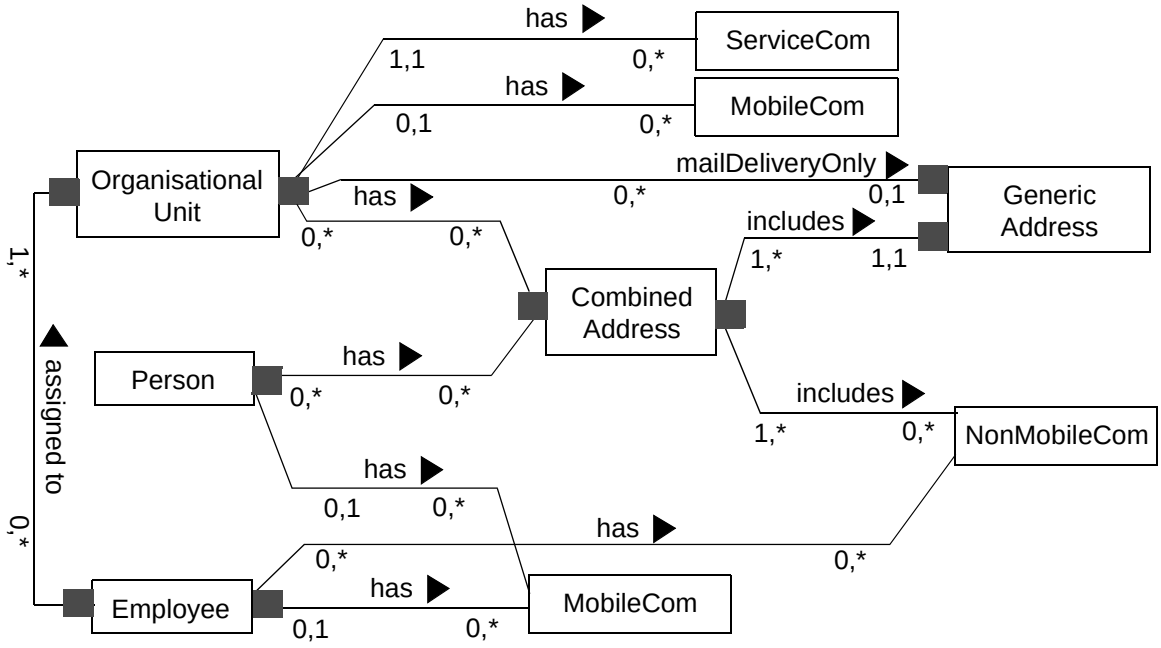

Fig. 20: Adding Object References

The following listing shows part of the Smalltalk code that has been generated by a CASE tool from this model. Since Smalltalk does not feature static typing, the attributes (instance variables) are not typed within the class declaration. Notice that on the code level, there is no difference between attributes (like number in GenericAddress or organisationalUnit in CombinedAddress) and associated objects any more. They are both implemented as references to objects which are stored in instance variables. The protocol of a class (the set of its signatures) is listed after the class definition - again without any type information about the parameters or returned objects.

```
Object subclass: #GenericAddress
    instanceVariableNames: 'number street '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Organisation'

GenericAddress selectors IdentitySet (#number: #number #street: #street)

number
    "RETURNS: String"
    "This method was generated on 2 September 1998 at 3:32:13 pm"

    ^number
```

number: aString

"*aString: String*
RETURNS: String"
"This method was generated on 2 September 1998 at 3:32:13 pm"

number := aString.
^aString

Object subclass: **#CombinedAddress**
instanceVariableNames: 'organisationalUnit '
classVariableNames: ''
poolDictionaries: ''
category: '*Organisation*'

CombinedAddress selectors IdentitySet (#organisationalUnit #protectedRemoveFromOrganisationalUnit: #addToOrganisationalUnit: #protectedAddToOrganisationalUnit: #removeFromOrganisationalUnit:)

protectedAddToOrganisationalUnit: anOrganisationalUnit

"*Establish the bidirectional association 'belongsTo' ('has') between the receiver and anOrganisationalUnit only from the receivers side.*
This method must only be called from instances of class OrganisationalUnit.
The method returns true if the association is established, false otherwise."
"This method was generated on 2 September 1998 at 3:32:13 pm"

anOrganisationalUnit isNil ifTrue: [^false].
organisationalUnit isNil ifTrue: [organisationalUnit := Set new].
(organisationalUnit includes: anOrganisationalUnit)
ifTrue: [^true].
organisationalUnit add: anOrganisationalUnit.
^true

Object subclass: #**OrganisationalUnit**
instanceVariableNames: 'combinedAddress employee serviceCom mobileCom '
classVariableNames: ''
poolDictionaries: ''
category: '*Organisation*'

OrganisationalUnit selectors IdentitySet (#addToEmployee: #protectedRemoveFromEmployee: #employee #protectedAddToCombinedAddress: #mobileCom #protectedAddToServiceCom: #protectedRemoveFromMobileCom: #protectedAddToEmployee: #addToMobileCom: #protectedAddToMobileCom: #removeFromMobileCom: #removeFromCombinedAddress: #removeFromServiceCom: #combinedAddress #addToCombinedAddress: #addToServiceCom: #serviceCom #protectedRemoveFromServiceCom: #removeFromEmployee: #protectedRemoveFromCombinedAddress:)

removeFromCombinedAddress: aCombinedAddress

"aCombinedAddress : CombinedAddress
RETURNS: nil or CombinedAddress

```
    Break the bidirectional association 'has' ('belongsTo') between the receiver and aCombinedAd-
dress.
    The method returns true if the association is broken, false otherwise."
    "This method was generated on 2 September 1998 at 3:32:13 pm"

    (combinedAddress includes: aCombinedAddress)
        ifFalse: [^aCombinedAddress].
    (aCombinedAddress protectedRemoveFromOrganisationalUnit: self)
        ifFalse: [^IWIError notify: #ProtectedCallFailed].
    combinedAddress remove: aCombinedAddress.
    ^aCombinedAddress
```

addToCombinedAddress: aCombinedAddress

```
    "aCombinedAddress : CombinedAddress
    RETURNS: nil or CombinedAddress

    Establish the bidirectional association 'has' ('belongsTo') between the receiver and
    aCombinedAddress.
    If the association is established the method returns the argument, otherwise an IWIError is raised."

    "This method was generated on 2 September 1998 at 3:32:13 pm"

    aCombinedAddress isNil ifTrue: [^IWIError notify: #NilNotAllowed].
    combinedAddress isNil ifTrue: [combinedAddress := Set new].
    (combinedAddress includes: aCombinedAddress)
        ifTrue: [^aCombinedAddress].
    (aCombinedAddress protectedAddToOrganisationalUnit: self)
        ifFalse: [^IWIError notify: #ProtectedCallFailed].
    combinedAddress add: aCombinedAddress.
    ^aCombinedAddress
```

4.2 Orders and Stock Management within a Wholesale Grocery Company

A wholesale grocery company offers many different types of goods. Typically, it keeps a large amount of units - like bottles, cans, boxes etc. - of every type in stock. At first sight it may appear as a good idea to map every real world object - like a unit of goods - to an object of the information system. However, such an approach is not satisfactory in the end. While the units that are stored and ordered certainly have their own identity simply by their physical existence, order processing and stock management will usually not differentiate between particular physical items. This is for a simple reason: All items of a certain type have the same properties. Therefore the effort to differentiate between particular items does not make much sense. We assume that every product in stock is stored in some kind of a container - like a can, a bottle, a box etc. A container itself can be part of another container. A particular product - like orange juice - may be available in different containers. Therefore we separate the description of a product from the description of a container. Notice, however, that certain features we normally associate with a container - like the name of the product it stores, the company that produces the item or the corresponding market segments - may be assigned to the product in order to avoid redundancy. Sometimes these features will have to be redefined for a specific container.

While it is not appropriate to regard a container as a specialisation of a product (an instance of product should exist independently of any corresponding containers), delegation allows to express the relationship between product and container in an adequate way. Therefore we regard Container, an abstraction of different container classes, as a role of Product. This implies that all the services offered by an instance of Product are transparently available to instances of concrete subclasses of Container. For example: If such an instance receives the message "name" and this message is not includes in its own protocol, it would dispatch it to its associated role filler. If the container is explicitly associated with companies that deliver it, this association would be used within the corresponding service "deliveredBy" instead of delegating it (in this case explicitly) to the role filler. The following method definition in Smalltalk illustrates the implementation of deliveredBy within Container:

deliveredBy

```
(self suppliers isEmpty) ifTrue: [^self roleFiller deliveredBy]; ifFalse: [^self suppliers]
```

Both, marketing research and stock management may require to assign products to categories. We assume that every product can be assigned to one category only. A category itself can be part of one superordinate category: orange juice as part of juices as part of beverages. Fig. 21 illustrates a preliminary object model that maps the concepts discussed so far. Notice that an instance of Container (or its subclasses respectively) is assigned to exactly one product. In principle, a container, like a bottle, could be used for a number of products. However, we want to use instances of Container to store the actual amount of units of certain products. Some containers, like bottles, do not contain any other container. Similar to categories and products we use an abstract class (Container) and a recursive association to express the difference between basic containers and aggregated containers. The recursive associations between Container and SuperContainer, as well as between AbstractProduct and Category are transitive. The easiest way to express this contraint is to use an aggregation association.

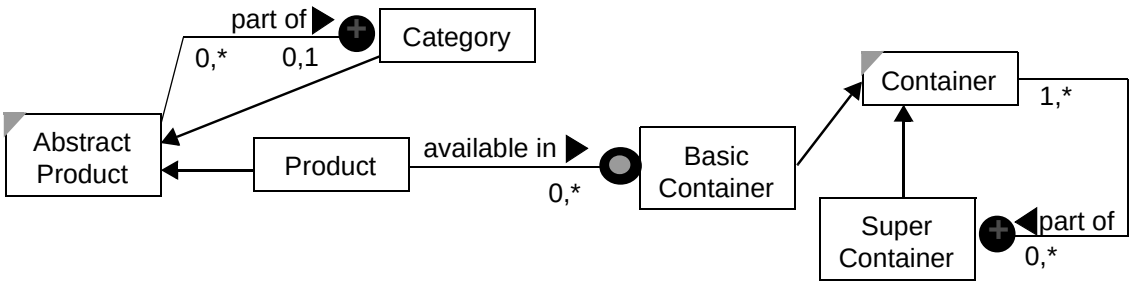

Fig. 21: Products and Containers

It is the main purpose of stock management to keep track of available amounts and to arrange for ordering additional amounts in time. Since we do not want to create an instance of Container for every real world container, we will use an instance to represent a type of container - like a bottle of a particular beverage. Information about the actual amount of containers or the minimum amount can be stored within the corresponding instance of Container. Containers may be part of other containers. Therefore we have to decide where to store amounts without producing redundancy. Usually, it should be best to store an amount on the highest level of aggregation. The amount of units on lower levels of aggregation can be calculated from that amount.

Sometimes, however, there are different containers of the same product in stock. For instance: A container that includes six six-packs of bottles has been opened to remove some six-packs. In this case there are containers that exist independently from their previous super container. In order to deal with this requirement, it is possible to store the amount of independent units on every level of aggregation. Additionally, there is a service that allows to calculate the number of units that are stored on a higher level of aggregation. The attribute minimumAmount is related to the number of units on the corresponding level.

Any BasicContainer is characterized by the unit of measurement that applies to its content (like litre for a bottle) and the amount of units. A SuperContainer has an attribute that stores the number of sub-containers it contains. Container includes an attribute to store the actual amount of independent instances and the minimum amount of instances. Additionally, each Container includes a service that computes the total number of instances - which is the number of independent instances plus the derived number of instances that are part of superordinate containers. While a particular container can be part of one subcontainer only, the multiplicity in the model expresses the fact that a type of container can be associated with many different types of super-containers, for instance: a bottle with a six-pack or directly with a box. Let us consider an example: A beverage is available in bottles (corresponds to instance of BasicContainer) which are aggregated to six-packs (corresponds to instance of SuperContainer). On the highest level of aggregation a box (corresponds to instance of SuperContainer) contains six six-packs. Assume there are 40 boxes available in stock that contain eight six-packs each. Also, there are three independent six-packs and five independent bottles. That would result in (40 * 8 + 3) * 6 + 5 = 1943 bottles.

In general, the actual amount of bottles would be calculated recursively by totalAvailableAmount within corresponding instances of BasicContainer or SuperContainer. For a particular instance of BasicContainer or SuperContainer this could be expressed by the following Smalltalk code for the method totalAvailableAmount within the class Container:

```
totalAvailableAmount

    self superContainer isNil ifTrue: [^self numberOfIndependentUnits].
    ^self superContainer totalAvailableAmount * self superContainer
    numberOfSubcontainers + self numberOfIndependentUnits
```

Notice that the code is simplified by the assumption that there is not more than one super-container associated with a container. Fig. 22 shows the structure of selected classes that are used for stock management.

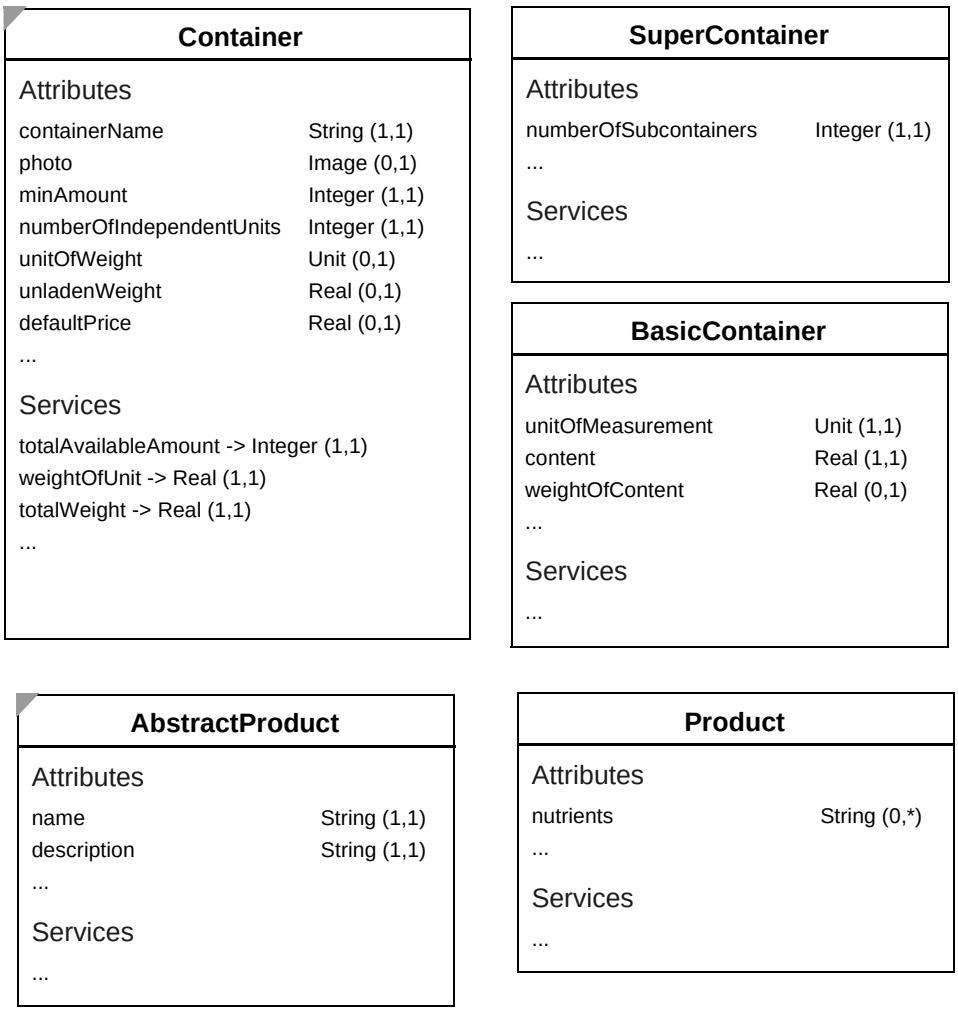

Fig. 22: Detailed Description of Products and Containers

Modelling the price of goods can be a peculiar problem. Let us assume that there is a number of different methods to determine the price of a unit which should be covered by the system:

1) add a certain percentage on the price the company has paid
2) ditto, however, possibly different for every market segment
3) have the corresponding sales manager determine the price which may be updated on request
4) ditto, however, possibly different for every market segment
5) in addition to 1) to 4): allow for discounts
6) allow sales personnel to negotiate prices with customers

We assume that the prices which are relevant for an order are the prices that are valid at the time of the order. Often, the price of an item is only a part of the overall sales contract which may also include agreements on cash discount, delivery, guarantee etc. In this case study we will look at the price only. Within the object model any of the methods listed above should be

taken into account. When it comes to select the method that is to be used within a certain context, the following rule would apply: A specific method overrules a more general one. For instance: If there is a price for every market segment, a product is offered in, and a price that had been negotiated with the customer, the "customized" price would be taken. Also, if there are prices assigned to containers of different levels of aggregation, an explicit price on a specific level overrules prices computed from prices assigned on a lower level.

The first calculation method seems to be trivial. However, it has to be specified which wholesale price should be used - the one paid for the latest delivery or the average price of all corresponding items in stock. To calculate an average wholesale price requires to take into account every single procurement of the corresponding container. It may be that do not feel in the position to make this decision forever. Fortunately, encapsulation allows us to abstract from possible future specifications: The actual semantics of the method used to calculate the relevant wholesale price is hidden behind a message selector which will be stable over time. In order to allow for prices that depend on context features (market segments, customers ...), it is necessary to introduce additional classes - like MarketSegment or SalesOrder. While other concepts are possible, we assume that every customer is assigned to exactly one market segment. A product can be assigned to many market segments. Sometimes it may be necessary to assign containers of a certain product to different market segments. In this case, the market segments assigned to a container "redefine" the market segments assigned to the corresponding product. Fig. 23 illustrates the object diagram used for price calculation. To simplify the model, we assume that a product is produced by exactly one company.

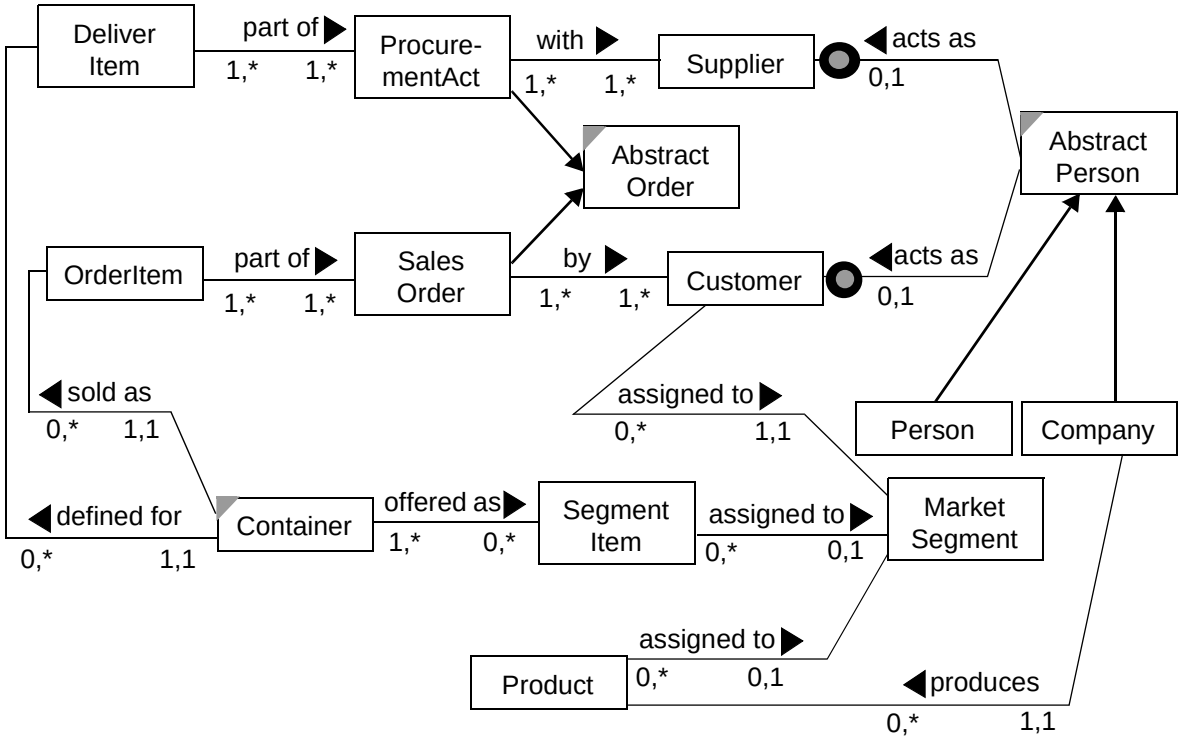

Fig. 23: Purchasing and Selling of Products/Containers

Every container is assigned a recommended sales price which is stored in the attribute defaultPrice. The price per unit which is used within a particular sales order can be obtained from pricePerUnit within OrderItem. If the corresponding attribute individualPricePerUnit is instantiated, pricePerUnit delivers that attribute's value. Otherwise the price it calculated from segmentPrice within the corresponding instance SegmentItem. This instance can be selected from all instances of SegmentItem which are associated with the container: It is the particular instance that is also associated with the market segment the customer is part of.

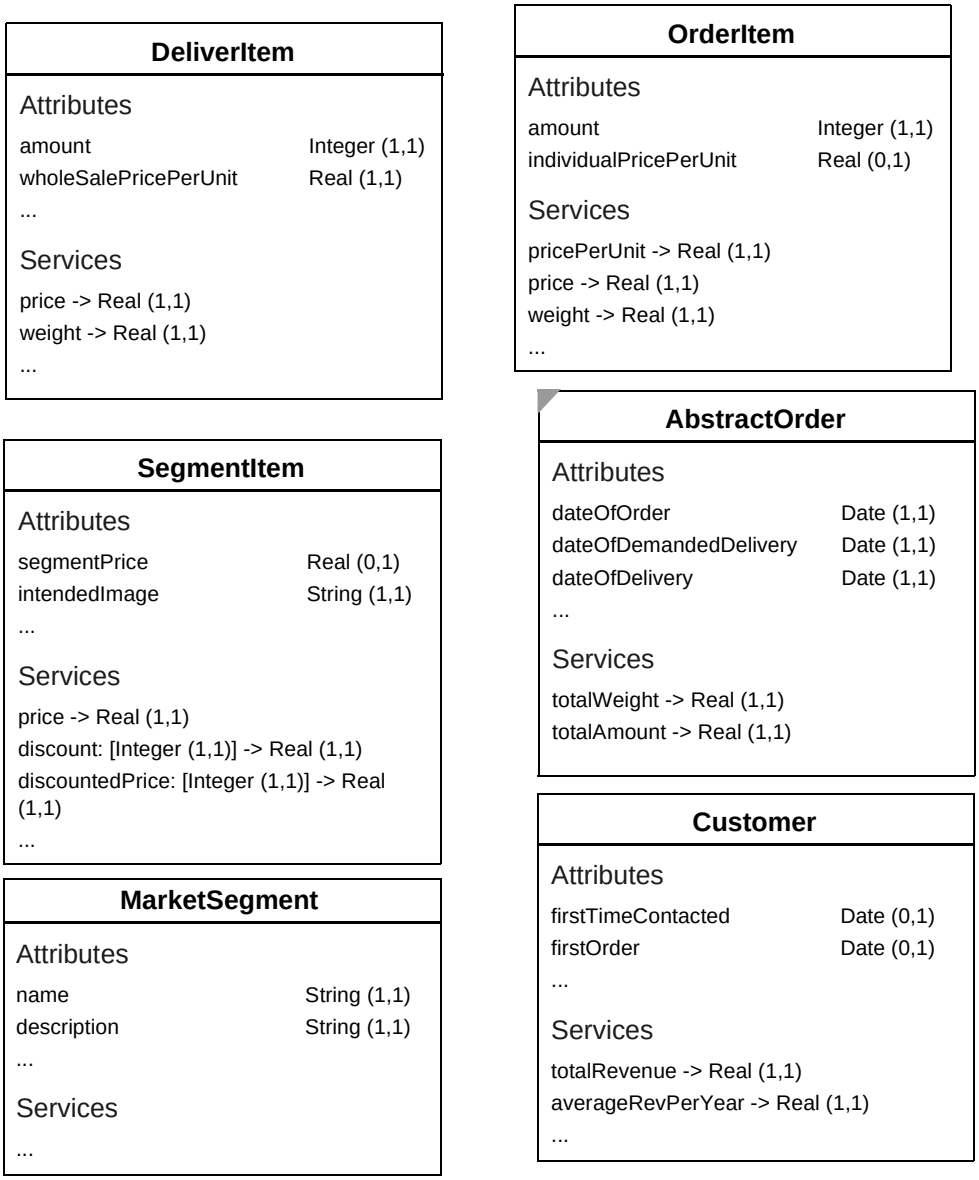

Fig. 24: Detailed Description of selected Classes

If a customer asks for the price he has to pay for a certain amount of a specific container, the clerk would first look at the price delivered by the service price of SegmentItem. If a specific price for this container had been defined for the container within the market segment the customer is part of, this price would be taken. Otherwise the service would refer to the service de-

faultPrice provided by the corresponding container. Discounts would be taken into account by applying the service discountedPrice: amount within SegmentItem within SegmentItem. Notice that we could introduce an abstract superclass of DeliverItem and OrderItem. In order to prepare for implementation, it is helpful to specify where to store references within associated objects. Fig. 25 shows the partial object model rendered in fig. 23 supplemented by the specification of object references.

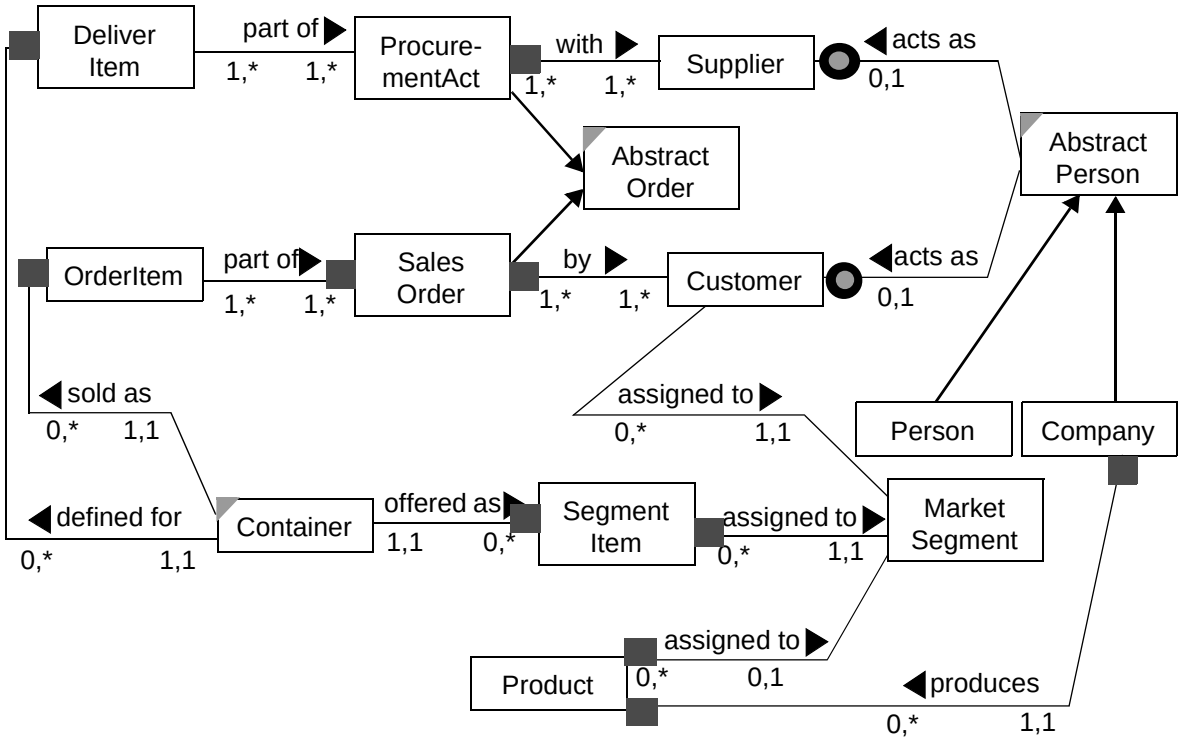

Fig. 25: Adding Object References

While the concepts for price calculation we have introduced in the example may appear complicated, one has to take into account that the variety of methods used to determine prices in practice is much higher. In addition to that, contracting may include other aspects like mode of payment and delivery etc.

4.3 Orders and Stock Management in a Car Dealership

Different from a wholesale grocery company, a car dealership sells items which are instances with an individual identity. Usually, a particular car has specific features (colour, equipment). In addition to that, a car has an individual history, part of which may be of interest to the car dealer. Therefore we model a car as an object. This approach, however, does not seem to be appropriate for spare parts. They are similar to the items sold in the wholesale grocery store. Therefore all spare parts of a particular type are represented by *one* object of the class SpareType. We differentiate three basic transactions with customers: selling a new car, selling a used car, providing service. The latter may include the sale of spare parts or additional equipment. Usually, new cars and used cars are handled within different business processes. Also, describing a used car requires additional information. At first sight, this may suggest to use to different

classes to model new and used cars. There is, however, one serious shortcoming with this approach: A particular new car will eventually become a used car. If we used two classes, an instance would have to migrate to another class during its lifetime. This will usually be accomplished by deleting the original instance, create a new one and copy the relevant state of the original instance to the newly created. We have already pointed to the problems that are caused by class migration (3.2). For this reason, we use a different approach: A new car is modelled as an instance of Car, while a used car is mapped to an instance of UsedCar which is a role of a corresponding instance of Car. Applying delegation allows to avoid the need for class migration. If a new car is to be recorded in the information system, one would create an instance of Car. If this car turns into a used car, one would simply create an instance of UsedCar and associate it with the instance of Car. Hence, the original instance of Car would not be deleted.

Depending on the detail of the description, modelling a car can be a very complicated matter. Not only that a car may consist of thousands of parts. Furthermore there are numerous constraints that apply to the combination of parts. For instance: tyres can be mounted to certain wheels only; a wooden steering wheel may not be available with an air-bag; an automatic transmission may not go with sports seats etc. Although this information is of vital importance both for selling and maintaining a car, it is not a good idea to include it entirely in a dealer's information system - basically for economic reasons: The manufacturer creates this information - and changes it over time. Therefore a dealer should rather access the information provided by the manufacturer instead of creating and maintaining his own. Nevertheless the dealer needs to store certain information about parts or extra equipment. Firstly, they may have to be expressed within contracts or invoices. Secondly, it may be relevant for marketing. For instance: At the beginning of summertime, the dealer wants to offer an air-condition to those customers who do not have one installed in their car. We assume that there is one sales contract for every car that is sold. In case a customer buys many cars at a time, there would be the need to handle many contracts. We also assume that exactly one sales representative is in charge of a particular sales contract. A car may either be bought or leased by a customer. Beside trading a car, there is also service/repair. Since these three types of transactions have a lot in common, we introduce the abstract superclass Contract as an abstraction. In case the car has been leased, the amount to be paid for service is either charged to the leasing firm or to the customer. In order to differentiate these two cases, we use to different classes to model leasing contracts, LeasingContract and FullServiceContract, the latter being a specialisation of the first. There are certain types of services, for instance: regular service, repair service. Every type can be assigned a description, like its name, the default price, the default duration etc. On the other hand there are concrete services. They are performed on a particular car by certain mechanic at a specific time. In order to avoid redundancy, we apply delegation: A concrete service is represented by an instance of ConcreteService which is a role of a corresponding instance of Service. This leads us to a first version of the object model (fig. 26).

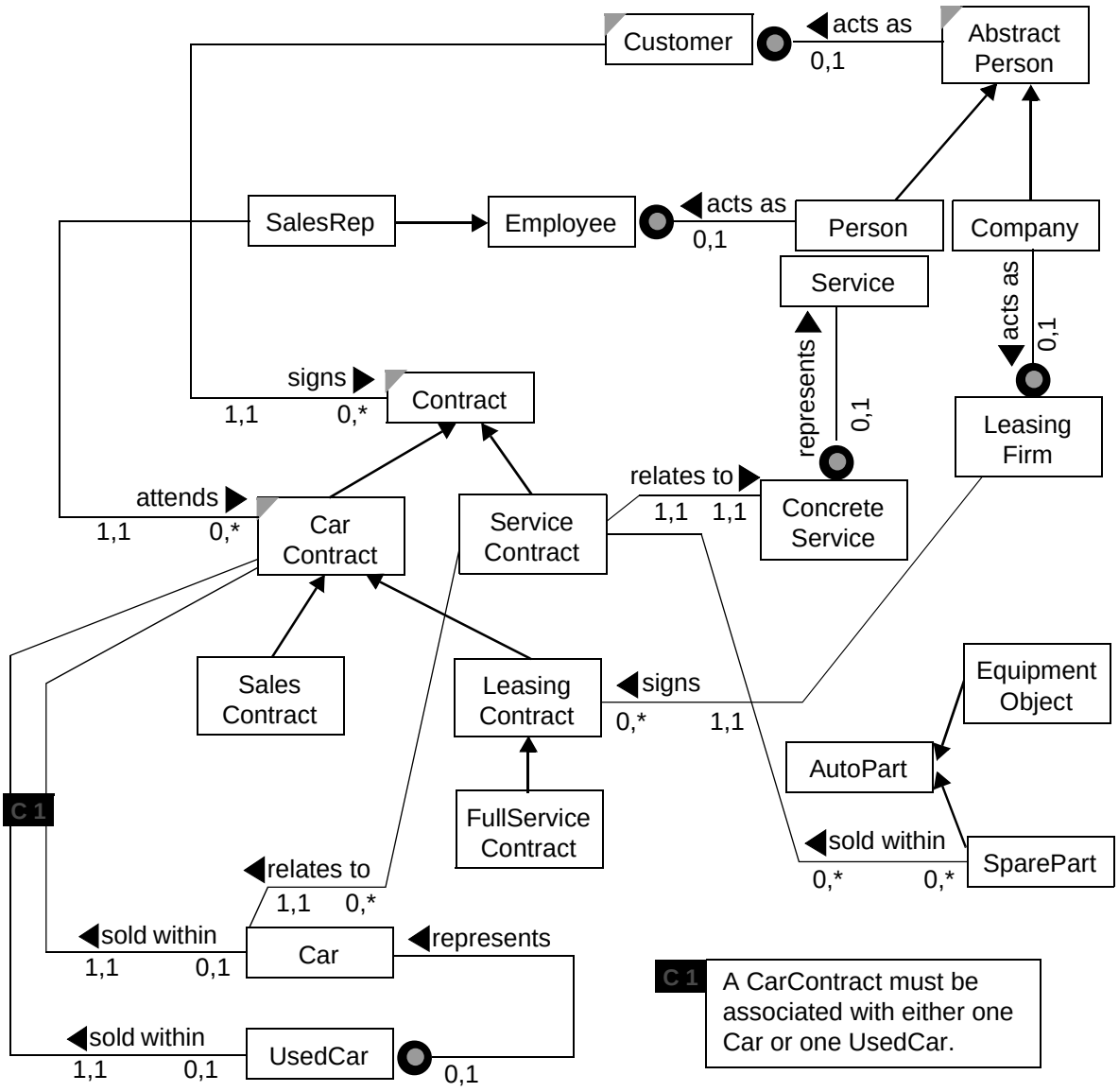

Fig. 26: Contracts, Products and Contractors

A particular car is of a certain type which in turn may be part of a series. For instance: A particular car may be of the type "316 i" of the series "BMW 3/B". Certain features of a car - like its colour or supplemental equipment - are specific for a particular instance. Other features - like standard equipment, engine, base price - are determined by its type. There are also features which do not differ within a whole series - like manufacturer, length, width, height. It seems to be intuitive to model a car as an instance of a type where a type is an instance of a series. However, such an approach would cause a problem: We would have to model a type as a metaclass and a series as a meta-metaclass. This is not feasible - not only because it would probably be confusing to many people, but also because there is not concept like a meta-metaclass available in MEMO-OML. If we look at a particular car, we are interested in its feature - usually abstracting from the fact whether it is an individual feature (like its colour) or a type fea-

ture (like its engine). However, it is not a good idea to store type specific information with an object that represents a particular car, since that would cause an extensive amount of redundancy. Therefore we introduce Car as a role of Type which in turn is a role of Series. If an instance of Car receives the message engine which is not included in its own interface, it would be transparently dispatched to the associated instance of Type or Series respectively.

In order to support the specification of valid sales contracts, the supplementary equipment is explicitly associated with every instance of Type and Series. In addition to that, the associations named "includes" specify the supplementary equipment that is included. The supplementary equipment assigned to a particular car has to be a subset of the superset of the supplementary equipment allowed for the corresponding type and series. Notice that we regard a colour (of paint or of seats) as an equipment object, too.

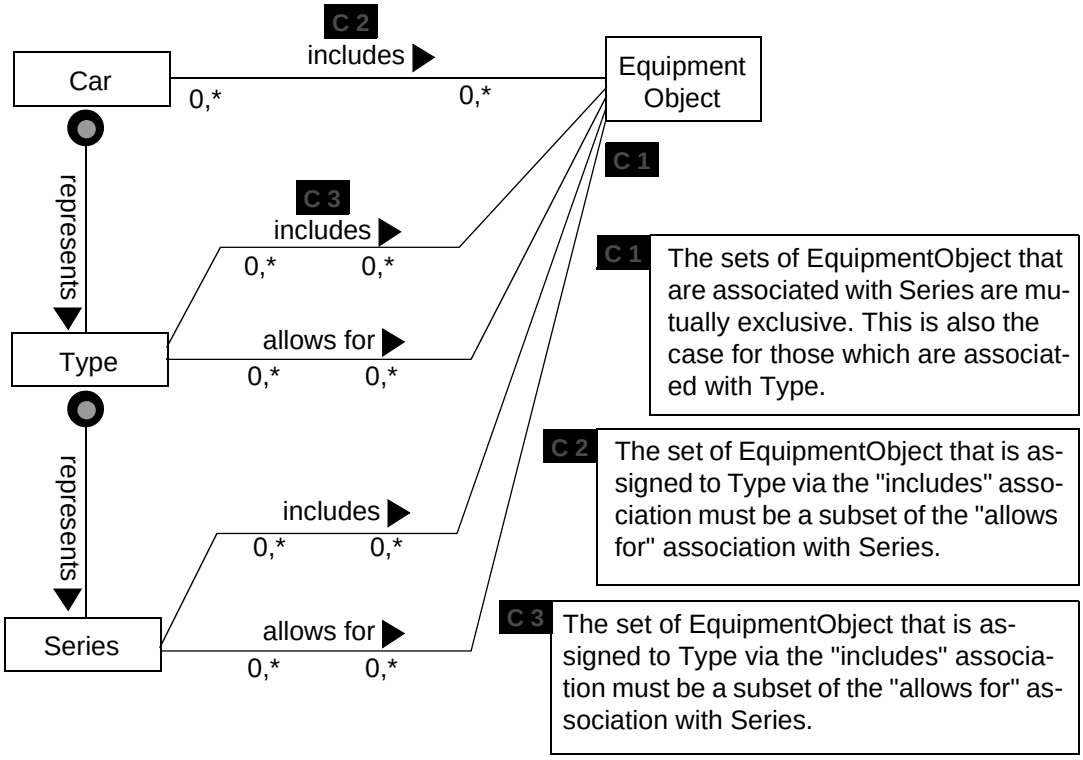

Fig. 27: Cars, Types and Series

While a type and a particular car are assigned a recommended sales price, it is possible to assign a special sales price to any sales contract. We assume that the price paid by the dealer is also a matter of individual assignment. We also assume that the dealer buys new cars from a manufacturer or another dealer (hence, from a company) only, while used cars can be bought either from a person or a company. Fig. 28 illustrates the classes used for taking into account the procurement of cars. This is certainly a simplified model because dealers will usually have more complicated contracts with car manufacturers.

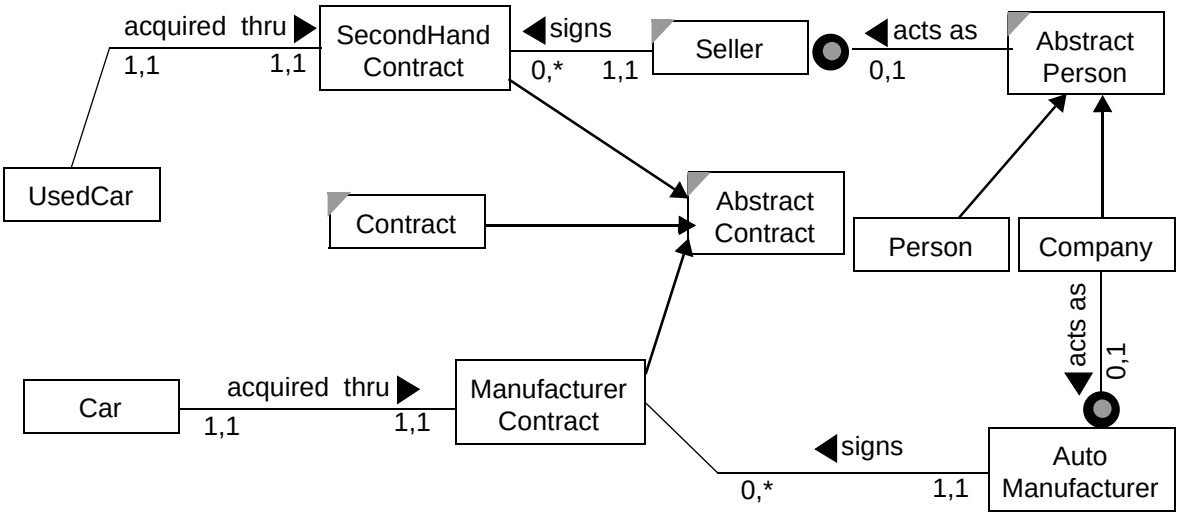

Fig. 28: Cars, Types and Series

In order to get a more detailed view of the model, we have to augment the classes with attributes and services. Fig. 29 and fig. 30 give a detailed description of selected classes.

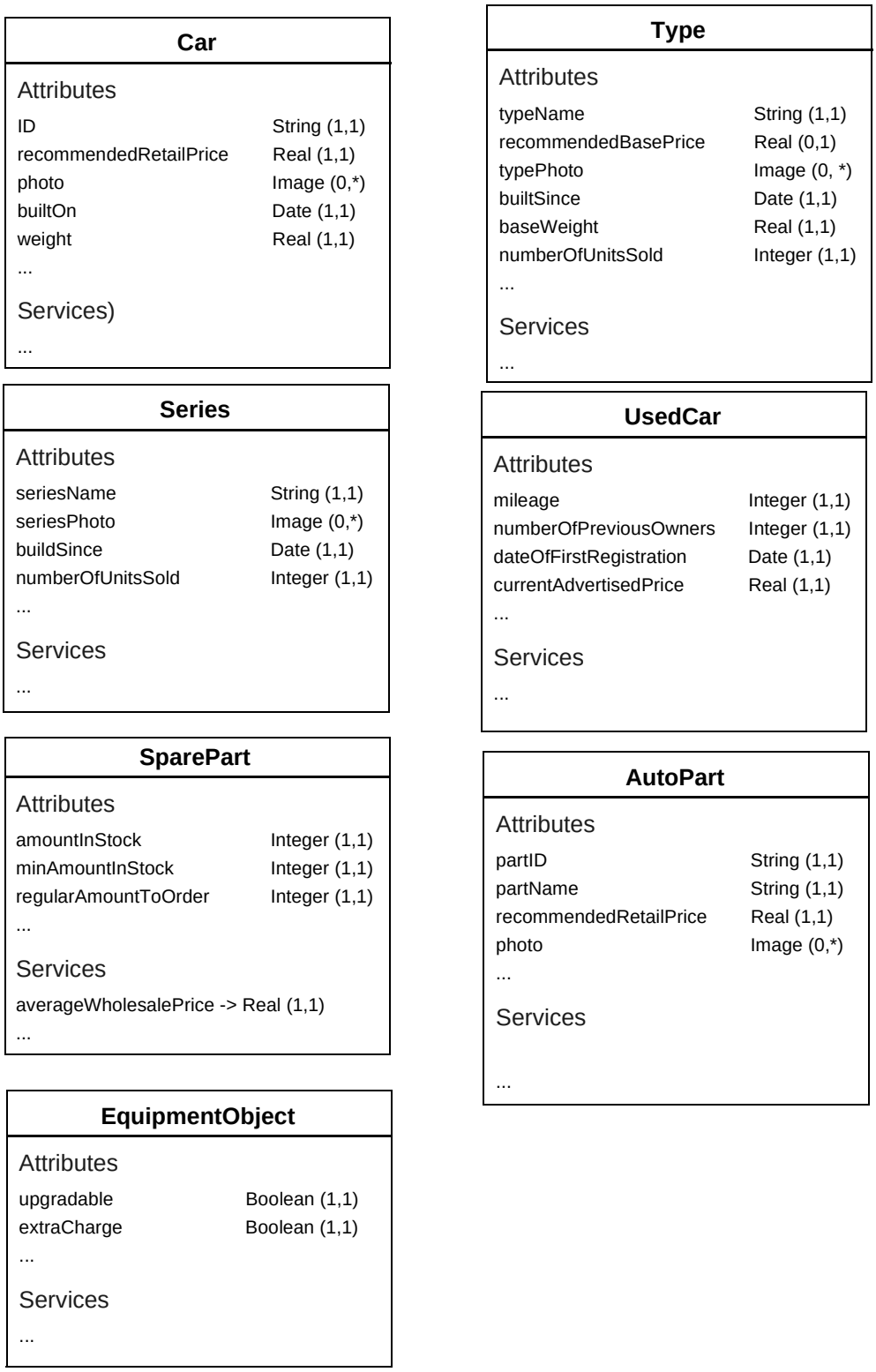

Fig. 29: Detailed Description of Cars and Parts

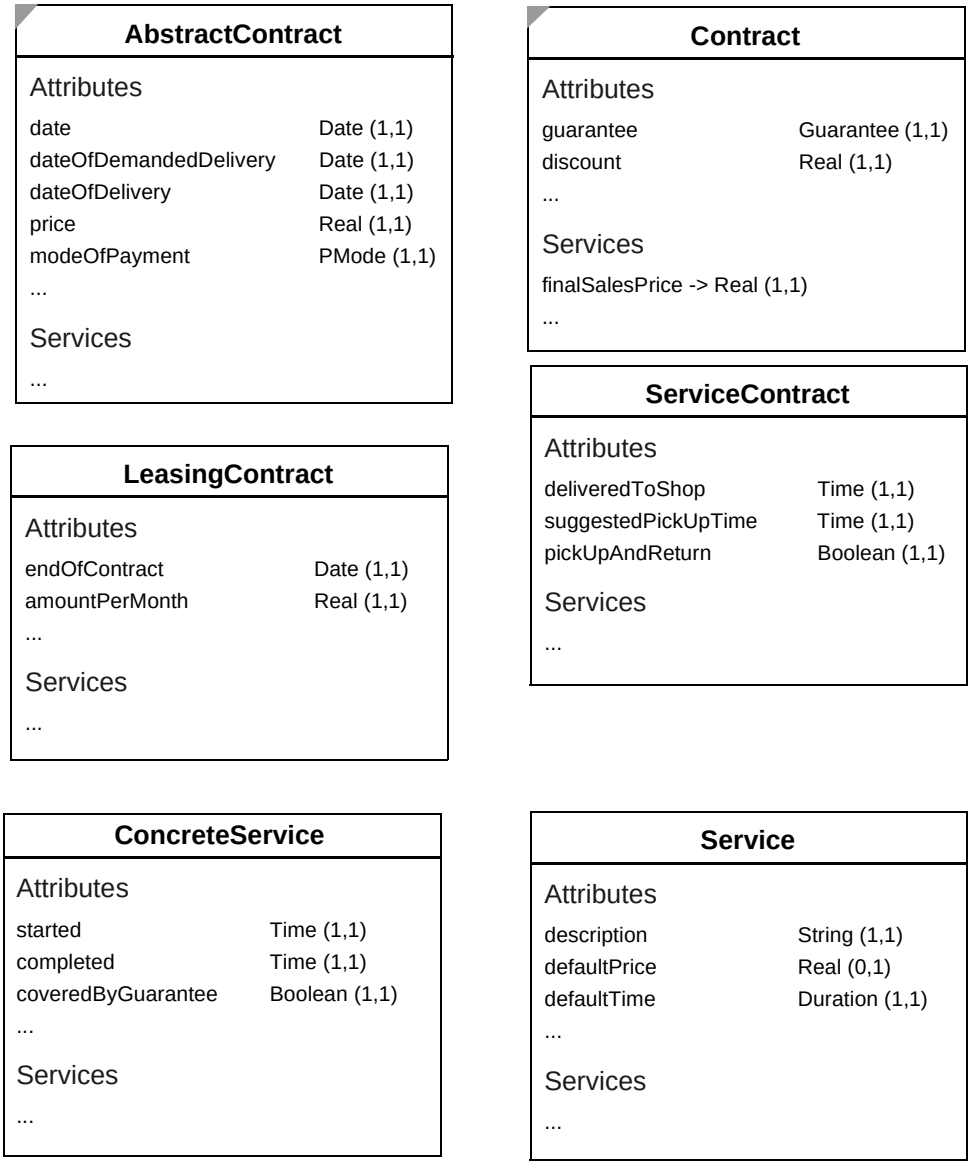

Fig. 30: Detailed Description of Services and selected Contracts

5. Concluding Remarks

Designing high quality object models of domains typical for corporate information systems is usually a challenging task. Not only that it is hard to find abstractions, many people can agree on - and that prove to be satisfactory over time. In addition to that, modelling in practice has to take into account the economical considerations. Sometimes it may be more economical to go with a less elaborated model. The economics of modelling projects, however, is hard to judge. In every single case there is need for a careful analysis about the adequate level of detail, formalisation and abstraction.

The small case studies presented in this report are by no means intended to reference models for certain domains. It should be clear that they are based on assumptions which reduce re-

markably the diversity and complexity of real world domains. The purpose of the examples and case studies is to provide a concrete subject for reflection and discussion about object-oriented modelling. Therefore the reader should try to understand and judge the examples - and to develop alternative abstractions. Modelling is a challenging intellectual task. While there are a number of design principles and heuristics, a great deal of the competence required seems to be an art. One part of this art relates to the intellectual ability to identify and evaluate abstractions for the purpose a model should serve. Developing models and analysing existing models helps to develop this kind of competence. The other part of the art implies social skills: Among other things, a model should provide a medium to foster communication. That recommends to use concepts and visual representations other participants are familiar with. Usually, one cannot expect that every part of a model is intuitive. Instead, the concepts of a model have to be explained to others and sometimes they will have to be revised. That requires to have an idea of other people's perception, of the language they speak - and also the willingness to revise his own beliefs. However, that does not mean that a model should always represent exactly the ideas of domain experts. Instead, a good designers has the responsibility to find a balance between the requirements explicitly stated by domain experts and the concepts he favours based on his own a rational analysis of the domain.

References

[Bun74] Bunge, M.: Treatise on Basic Philosophy. Vol. 1. Dordrecht, Boston: Reidel 1974

[Fra98a] Frank, U.: The MEMO Meta-Metamodel. Arbeitsberichte des Instituts für Wirtschaftsinformatk, Nr. 9, Koblenz 1997

[Fra98b] Frank, U.: The MEMO Object Modelling Language (MEMO-OML). Arbeitsberichte des Instituts für Wirtschaftsinformatk, Nr. 10, Koblenz 1997

[JaCh92] Jacobson, I.; Christerson, M; Jonsson, P; Overgaard, G.: Object-Oriented Engineering. A Use Case Driven Approach. Reading, Mass.: Addison-Wesley 1992

[KrLi95] Krogstie, J.; Lindland, O.I.; Sindre, G.: Towards a Deeper Understanding of Quality in Requirements Engineering. In: Iivari, K.; Lyytinen, K.; Rossi, M. (Eds.): Proceedings of the 7th Conference on Advanced Information Systems Engineering (CaiSE '95). Berlin et al.: Springer 1995, pp. 82-95

[Lin94] Lindland, O.I.; Sindre, G.; Sølvberg, A.: Understanding the Quality in Conceptual Modeling. In: IEEE Software, vol. 11, no. 2, 1995, pp. 82-95

[MoSh94] Moody, D.L.; Shanks, S.: What Makes a Good Data Model? Evaluating the Quality of Entity Relationship Models. In: Loucopoulos, P. (Eds.): Entity-Relationship Approach - ER'94. Business Modelling and Re-Engineering. 13th International Conference on the Entity-Relationship Approach. Berlin, Heidelberg etc.: Springer 1994, pp. 94-111

[Wie95] Wieringa R.J., Jonge W. de, Spruit P.A.: Using Dynamic Classes and Role Classes to Model Object Migration. In: Theory and Practice of Object Systems, 1, 1995, pp. 61-83