Published in: Journal of Object-Oriented Programming. Vol. 13, No. 3, June 2000, pp. 13-18

Delegation: An Important Concept for the Appropriate Design of Object Models

Ulrich Frank
University of Koblenz
Ulrich.frank@uni-koblenz.de

Abstract

In many application domains, there are certain aspects that cannot be modeled in an adequate way by using generalization - although it may be the concept of choice at first sight. Sometimes common associations such as interaction or aggregation will fail as well. In those cases, *delegation* often proves to allow for an appropriate abstraction. While delegation has been an important concept in different areas of computer science (mainly within AI and programming languages) for a long time, it is not explicitly offered by any of the major object-oriented modeling languages. In this paper, we introduce a concept of delegation as part of an object-oriented modeling language. First, we will analyze why both inheritance and common associations sometimes fail to model certain aspects of the real world. Against this background, it will be shown how delegation allows to fill this conceptual gap. The semantics of delegation as a modeling concept is specified in a metamodel. In order to foster the appropriate use of delegation, we provide a few examples together with a number of general design criteria.

1. Introduction

Within various projects, we had to realize that often neither inheritance nor commonly used associations (like interaction or aggregation) seemed to be appropriate concepts to model certain aspects of the real world. Instead, delegation proved to fill this conceptual gap in many cases. Delegation has been an important concept in different areas of computer science (mainly within AI and programming languages). In conceptual data modeling, the need for delegation was emphasized long ago [BaDa77]. In various publications on object-oriented software development delegation is mentioned as well ([Rum93], [GoRu95], [KaSc96], [Lie86], [BaDo96], [Sci89], [IBM94], [Wie95]). However, none of the major object-oriented modeling methods (such as [Boo94], [Jac92], [Rum93]) includes delegation explicitly as a concept of its own. This is the case for the current version of the UML [OMG99], too. We presume that this is for a number of reasons: Overestimation of the expressive power of inheritance, the ambiguity of inheritance, and the fact that most object-oriented programming languages do not allow for a convenient and safe implementation of delegation. We will first analyze the conceptual shortcomings of both inheritance and common associations. Then we will introduce delegation as a concept for object-oriented modeling.

2. Limits of Inheritance

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, because 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. 1).

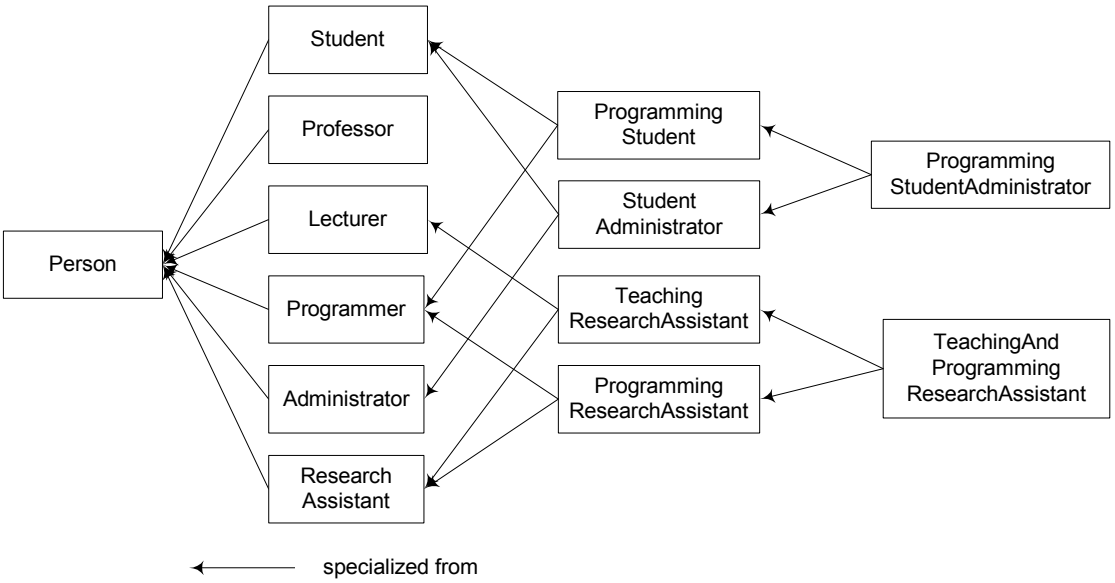

Fig. 1: Concepts resulting from Multiple Inheritance

Classes defined in this hierarchy would in principle allow the expression of 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 and as a student in another context. With most object-oriented programming languages, inheritance is specified in a way that, in our case, it 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. A 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 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 us to "freeze" certain abstractions before having instantiated a single object, while we sometimes need concepts that allow us to change abstractions after objects have been instantiated.

3. Alternatives to Inheritance

It is surprising that the problem we have discussed so far is hard to find in publications on object-oriented modeling. Some of the rare examples are [IBM94] and [Rum93].

3.1 Interaction

[IBM94] outlines the example of on object model for an auction. Among the classes the authors identify are Person, Auctioneer, Bidder, and Seller. They explicitly advise against the use of inheritance: "This is because it is possible for the same person to be a bidder, an auctioneer, and a seller." ([IBM94], p. 140). Instead they use an "interaction"-association, indicating that - for example - an instance of the class Bidder uses an instance of the class Person. Fig. 2 shows how to model our example domain with interaction associations. This approach helps to avoid redundancy, and adds flexibility to our model as well. However, it has one severe disadvantage: By treating those special associations like any other interaction association, we completely neglect the semantics that is characteristic for these associations in the real world: A bidder is a person. In other words: We would know more about our domain than we could express in our model - although this knowledge would be relevant for system implementation.

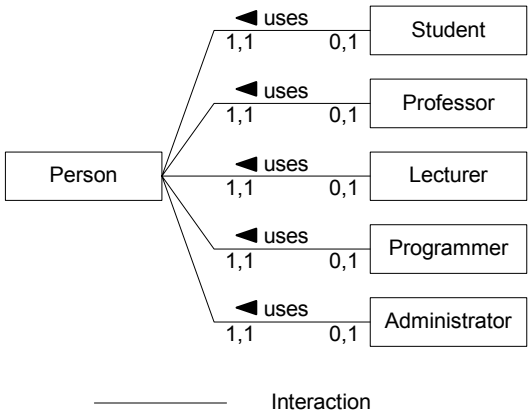

Fig. 2: Modeling the Example with Interaction Associations

3.2 Aggregation

Rumbaugh et al. suggest the use of "delegation" which they define as "aggregation of roles" ([Rum93], p. 67). In our example domain, they would regard a person as an aggregation of his appearances - we could also say: his roles - in various contexts. Firesmith et al. mention only briefly that aggregation could serve to provide some sort of delegation: "... the parts are visible to the aggregate and the aggregate can therefore delegate some of its responsibilities to its parts ..." ([FiHe96], p. 76.

Aggregation does not suffer from the problems we encountered for inheritance. Like in any other approach, the essential features of a person are specified in the class Person. The partic-

ular properties of a specific person (like his name, sex, etc.) are stored only within an instance of this class. Other features that may be relevant in certain contexts (like features of a student or a lecturer) are stored in role objects. Each role object would be regarded as a part of a corresponding instance of Person. A particular role object would not need to include any state that is managed within its associated object of the class Person. Hence, redundancy could be avoided. However, aggregation, too, is certainly not a satisfactory solution. This is for two reasons:

- There is a conflict with the common notion of aggregation. The semantics of aggregation is a delicate subject. None of the well-known methods or languages for object-oriented design (like [Rum93], [Boo94], [Jac92], [BoRu96]) provides a sound definition. Nevertheless, aggregation usually implies a notion of containment or even physical containment. We doubt that it is common sense to regard a role as a contained part of a person.
- Treating roles like other aggregated parts fails to express the special semantics we usually associate with roles. While we expect a person that performs a certain role (like a student) to still act like a person, this is certainly not the default for aggregates: You do not expect a wheel to act like a car. Therefore, similar to interaction, aggregation would force us to abstract from relevant semantics.

We can summarize that by no means aggregation provides a natural conceptualization of our example domain. Instead we find it to be a rather bizarre abstraction. What we are looking for is a special association that allows to express the semantics we have identified. For instance:

This association should imply that an object of the class Lecturer would behave like an object of the class Person. In order to avoid the confusion resulting from the ambiguity of "is a" we suggest to use other designators to characterize this sort of association. Instead of stating "a programmer is a person", we would rather say "a programmer *represents* a person" (or "a person *acts as* a programmer"). A programmer would then be regarded as a *role*. Different from inheritance, a particular instance of the class Person would propagate its state and behaviour to an instance of the (role-) class Programmer.

4. Delegation

There are two different perspectives on delegation, which are not always differentiated: an implementation or run-time point of view, and a conceptual point of view. On the implementation level, there is a remarkable amount of work on languages which feature delegation. In his classification of object-centered programming languages, Wegner calls languages which allow for inheritance *object-based*, while languages that support delegation *instead* are characterized as "classless objects with delegation" [Weg87]. Classless programming languages are sometimes called *delegation based*, while objects within these languages are called prototypical objects ([Lie86], [UnSm87]). For a detailed analysis of delegation based languages see [Mal95].

In most cases, delegation seems to be used with a programmer's perspective in mind: An object that receives a message which is not included in its own protocol *delegates* this message to another object (see, for instance ([GoRu95], p. 507). On a conceptual level, however, this point of view seems to be misleading: We would hardly say that a programmer delegates to a person when he is asked his name. Instead, we would rather say that a person delegates his responsibilities to roles that may represent him in specific contexts. Not only that implementation and conceptual level are usually not clearly differentiated, furthermore, there are alternative terms: Sciore uses "object specialisation" [Sci89] in order to express that

a "specialized" object "inherits" behavior from another object it can delegate messages to. Within the programming language Self, the object a message can be delegated to is called "parent object" [UnSm87]. Kappel and Schrefl introduce an association that they call "roleOf" ([KaSc96], pp. 32). Among other things, they characterize a "roleOf"-association by the notion of "Instanzvererbung" ("instance level inheritance").

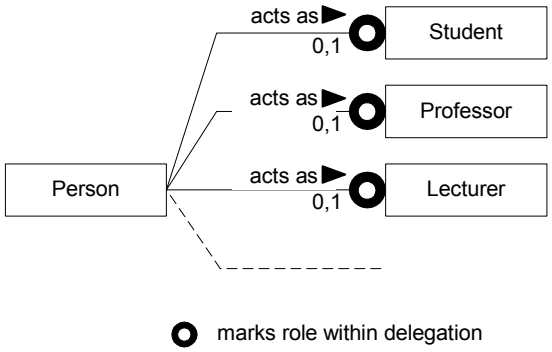

Fig. 3: Modeling the Example with Delegation

4.1 Semantics

Since our emphasis is definitely on the conceptual level, we prefer to speak of a responsibility delegated from a "delegator" to a "delegate". However, in order to avoid the ambiguity that might be caused by the fact that the term delegation is sometimes used in the opposite direction, we decided to use "role" and "role filler" instead. 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.
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 role 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 on does not allow for multiple delegation, since we regard integrity a more valuable asset than flexibility in this case. That does not necessarily exclude a role being 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.

Delegation is a concept provided by MEMO-OML, an object-oriented modeling language that is part of a method for enterprise modeling ([Fra97], [Fra98]). In order to bridge the semantic gap between a modeling language that includes delegation and an object-oriented programming language, we enhanced Smalltalk with delegation. The implementation of the delegation framework (fig. 4) is essentially based on overriding the "doesNotUnderstand" method common to all Smalltalk classes within an abstract class Role. RoleFillerModel and RoleModel, specialized from RoleFiller and Role, allow for transparent management (registration, notification) of dependants - similar to the behaviour provided by the Smalltalk class Model. By adding two alternative classes we leave it up to the user of the framework whether he wants to use the dependence mechanism or not: In case it is required to notify a role's dependants about its role filler's changes, you would specialize from RoleFillerModel and RoleModel, otherwise you would use the framework by defining subclasses from RoleFiller and Role (for details see [FrHa97]). The delegation framework is available on the web (www.uni-koblenz.de/~iwi/mobis/smalltalk). Notice, however, that with languages, which do not support dynamic typing, delegation cannot be implemented in such an elegant and flexible way.

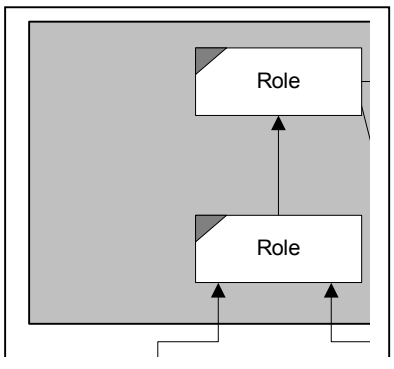

Fig. 4: The Delegation Framework and its Use via Specialization

To use delegation within a Smalltalk program, you specialize classes from the abstract classes provided by the framework. The objects instantiated from the specialized Role classes have to be registered with their corresponding role filler object. Nothing more is required. After that, each message that a role object does not understand is dispatched transparently to the associated role filler object. Executing the following code would result in executing the method firstName of the Person object.

```
| aPerson aStudent |

aPerson := Person new.
aPerson firstName: 'James'.
aStudent := Student new.
aPerson addRole: aStudent.      "Register aStudent as a role of aPerson"
aStudent firstName
```

4.2 Examples

The following two examples serve to illustrate typical cases for delegation. They are rendered using the graphical notation suggested by MEMO-OML ([Fra98]). More examples are presented in [FrHa97].

a) "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. 5). 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.

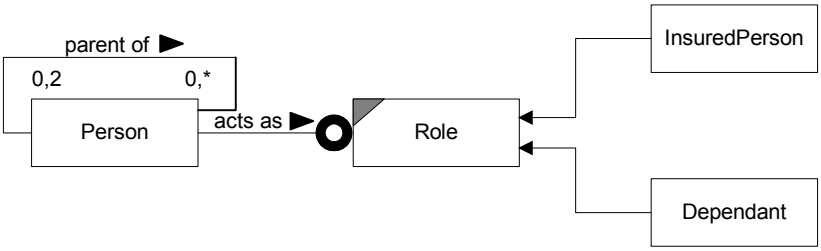

Fig. 5: Avoiding Class Migration through Delegation

b) "Multiple" role filler classes

A retail company serves both individuals and companies. Some of those companies also act as suppliers. 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 #3. 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 checking 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.

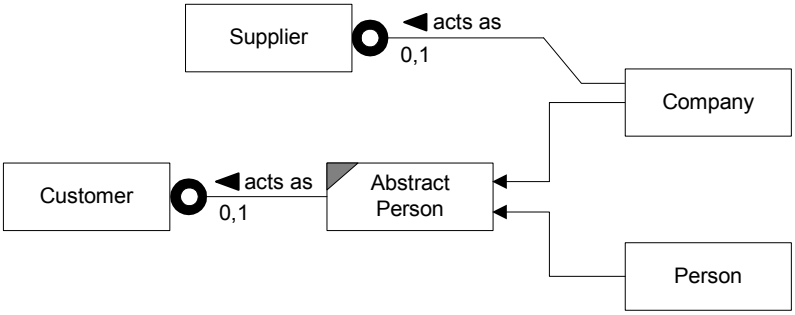

Fig. 6: 'Multiple' Delegation through Generalization

4.3 Guidelines for the Use of Delegation

While delegation can be a valuable alternative to inheritance, it is definitely not suited to replace it in general. In order to support the decision whether or nor to use inheritance, Rumbaugh et al. suggest focusing on the "essence" of inheritance: "Inheritance should only be used when the generalization relationship is semantically valid. Inheritance means that each instance of a subclass truly is an instance of the superclass; thus all operations and attributes of the superclass must uniformly apply to the subclass." ([Rum93], p. 284) In our opinion, this criterion is rather confusing - especially because Rumbaugh et al. refer to a different concept of inheritance than the one they use when they suggest aggregation as a concept to represent delegation (see 3.2). Applying Rumbaugh et al.'s suggestion to our first example would result in specifying the class Student as a subclass of the class Person - which is - taking into account the semantics of inheritance usually provided by programming languages - exactly what we wanted to avoid. While we do not agree to the rule of thumb Rumbaugh et al. suggest, we do not agree with Veryard either: "There are no fixed guidelines when to use

subtyping and when to use role entities; it is largely a matter of taste and style." ([Ver92], p. 54). Whether or not to apply delegation should always be based on a thorough analysis of the specific domain. We suggest a few guidelines that may help with this analysis.

- 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. Notions such as „task“, „job“, „serves as“, „works as“, etc may indicate the existence of a role. Therefore you should look for corresponding terms within available descriptions of a domain.
- A generalization that does not necessarily hold for the entire lifetime of the system to be designed could be a case for delegation. For instance: If a professor does not necessarily have to be an employee, 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.

Multi-level delegation should be used with specific care. While it may provide a more natural abstraction of certain real world aspects, it can make it more difficult to debug and maintain code. In general, one should beware of exaggerating the use of delegation.

5. Concluding Remarks

Delegation is an important concept to enrich both conceptual models and languages used on the implementation level. The advantages it offers are certainly not only of academic relevance. Kathuria/Subramaniam, who suggest a similar concept that they call "assimilation" (however, not as part of a modeling language), state: "As practitioners we have a strong need for a concept like assimilation." ([KaSu96], p. 39). While delegation has been subject of numerous publications in the area of object-oriented programming languages ([BaDo96], [Lie86], [Sci89], [Ste87], [Weg87]), popular object-oriented modeling methods (like [Boo94], [Jac92], [Rum93]) do not include it as concept of its own. This is also true for recent efforts to suggest "unified" or "open" (and eventually standardized) modeling languages (like [FiHe96], [OMG99]).

One essential motivation to introduce delegation is to be seen in the shortcomings of inheritance to model certain aspects of the real world. However, the semantics of inheritance is not specified in a unique way - if it is specified at all. The concept of inheritance we use is adapted from common object-oriented programming languages such as Smalltalk, Eiffel, C++, etc. This concept is different from a notion of inheritance that is known as "set-oriented" or "extensional". It is based on the idea of a class as a set of objects with common features. This set can be divided into subsets, hence subclasses, each of which is characterized by additional features. Different from intentional inheritance, each instance of a class is an instance of its superclasses at the same time (for a comprehensive comparison see [KaSc96], pp. 15).

Extensional inheritance provides features that are very similar to delegation. It can be

expected that the next generation of mainstream data base systems ([StMo95]) will support extensional inheritance. At first sight, this perspective may suggest that delegation will eventually become obsolete. However, it might even promote the future importance of delegation. Mainstream object-oriented programming languages do not support extensional inheritance. Moreover, there is a good reason why this will not change in future times: It is an essential concept of those languages that an object is an instance of exactly one class, not of many classes, as it would be the case with extensional inheritance. Hence, a mismatch can be expected between the concepts of inheritance used in programming languages and in some future database management systems. Delegation could serve to overcome this problem: Although both concepts do not offer identical semantics, delegation could be an option to represent extensional inheritance.

References

[BaDa77] Bachman, C.W.; Daya, M.: The role concept in data models. In: Proceedings of the 3rd International Conference on Very Large Databases 1977, pp. 464-476

[BaDo96] Bardou, D.; Dony, C.: Split Objects: a Disciplined Use of Delegation within Objects. In: Proceedings of the OOPSLA'96. New York: ACM 1996, pp. 122-137

[Boo94] Booch, G.: Object-Oriented Analysis and Design with Applications. 2nd Ed., Redwood City: Benjamin Cummings 1994

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

[Fra94] Frank, U.: MEMO: A Tool Supported Methodology for Analyzing and (Re-) Designing Business Information Systems. In: Ege, R.; Singh, M.; Meyer, B. (Eds.): Technology of Object-Oriented Languages and Systems. Englewood Cliffs/NJ: Prentice Hall 1994, pp. 367-380

[Fra97] Frank, U.: Enriching Object-Oriented Methods with Domain Specific Knowledge: Outline of a Method for Enterprise Modelling. Arbeitsberichte des Instituts für Wirtschaftsinformatik. No. 4, Koblenz 1997

[Fra98] Frank, U.: The MEMO Object Modelling Language (MEMO-OML). Arbeitsberichte des Instituts für Wirtschaftsinformatik. No. 9, Koblenz 1998

[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

[GoRu95] Goldberg, A.; Rubin, K.S.: Succeeding with Objects. Decision Frameworks for Project Management. Reading/Mass. etc.: Addison-Wesley 1995

[How95] Howard, T.: The Smalltalk Developer's Guide to VisualWorks. New York: SIGS Books 1995

[IBM94] IBM: Introduction to OOP and IBM Smalltalk. IBM 1994

[Jac92] Jacobson, I.; Christerson, M.; Jonsson, P.; Overgaard, G.: Object-Oriented Engineering. A Use Case Driven Approach. Reading/Mass.: Addison-Wesley 1992

[JoZw94] Johnson, R.E.; Zweig, J.: Delegation in C++. In: Journal of Object-Oriented Programming. Vol. 4, No. 11, pp. 22-35

[KaSc96] Kappel, G.; Schrefl, M.: Objektorientierte Informationssysteme. Konzepte, Darstellungsmittel, Methoden. Wien, New York: Springer 1996

[KaSu96] Kathuria, R.; Subramaniam, V.: Assimilation: A New and Necessary Concept for an Object Model. REPORT ON OBJECT ANALYSIS & DESIGN, Vol. 2, No. 5,

1996, pp. 36-39
[Lie86] Lieberman, H.: Using prototypical objects to implement shared behavior in object-oriented systems. In: OOPSLA, 1986, pp. 214-223
[Mal95] Malenfant, J.: On the Semantic Diversity of Delegation-Based Programming Languages. In: Proceedings of the OOPSLA95. New York: ACM 1995, pp. 215-230
[OMG99] OMG: Unified Modeling Language Specification. Version 1.3, June 1999
[Rat97] Rational: OCL. Version 1.1. 09/01/1997 (http://www.rational.com)
[Rum93] Rumbaugh, J. et al.: Object Oriented Modeling and Design. Englewood Cliffs/NJ: Prentice Hall 1993
[Sci89] Sciore E.: Object specialization. In: ACM Transactions on Office Information Systems, Vol. 7, No. 2, April 1989, pp. 103-122
[Ste87] Stein, L. A.: Delegation is Inheritance. In: Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications, Orlando, Florida, October, 1987, pp. 138-146
[StMo95] Stonebraker, M.; Moore, D.: Object-Relational DBMSs: The Next Great Wave. San Francisco: Morgan Kaufmann 1995
[UnSm87] Ungar, D.; Smith, R.B.: Self: The Power of Simplicity. In: OOPSLA '87 Conference Proceedings. ACM Sigplan Notices Vol. 22, No. 12, 1987, pp. 227-241
[Ver92] Veryard, R.: Information Modelling. Practical Guidance. New York, London etc.: Prentice Hall 1992
[Weg87] Wegner, P.: Dimensions of Object-Oriented Language Design. In: Proceedings of the OOPSLA87. 1987, pp. 168-182
[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