UNIVERSITÄT
KOBLENZ · LANDAU

Institut für
Wirtschaftsinformatik

Fachbereich Informatik
Universität Koblenz-Landau

ULRICH FRANK
SÖREN HALTER

ENHANCING OBJECT-ORIENTED SOFTWARE DEVELOPMENT WITH DELEGATION

Januar 1997

Arbeitsberichte des Instituts für Wirtschaftsinformatik Nr. 2

UNIVERSITÄT
KOBLENZ · LANDAU

Institut für
Wirtschaftsinformatik

Fachbereich Informatik
Universität Koblenz-Landau

ULRICH FRANK
SÖREN HALTER

ENHANCING OBJECT-ORIENTED SOFTWARE DEVELOPMENT WITH DELEGATION

Januar 1997

Arbeitsberichte des Instituts für Wirtschaftsinformatik Nr. 2

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 der Verfasser/ Address of the authors:

Prof. Dr. Ulrich Frank
Dipl. Inf. Sören Halter
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
Prof. Dr. Stefan Klein

Bezugsquelle / Source of Supply:

Institut für Wirtschaftsinformatik
Universität Koblenz-Landau
Rheinau 1
56075 Koblenz

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

Abstract

In many application domains there are certain aspects that cannot be modelled in an adequate way by using generalisation or common associations (like interaction or aggregation). In those cases *delegation* often proves to fill this conceptual gap. While delegation has been an important concept in different areas of computer science (mainly within AI) for long, it is not explicitly offered by any of the major object-oriented modelling methods. The following report will introduce a concept of delegation as part of an object-oriented modelling method. First we will analyse why both inheritance and common associations sometimes fail to model certain aspects of the real world. In order to foster the appropriate use of delegation we provide a number of examples together with a checklist. While delegation is primarily a modelling concept it is desirable to have it in place on the implementation level as well. Otherwise one has to deal with the impacts of a semantic gap between model and code. In order to avoid such a gap we suggest a modification of Smalltalk that allows to use delegation in a rather transparent way. Finally delegation is documented as a design pattern.

1. Motivation

Within various projects we found 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). In various publications on object-oriented software development delegation is mentioned as well ([Rum93], [GoRu95], [KaSc96], [Lie86], [BaDo96], [Scio89], [IBM94]). However, none of the major object-oriented modelling methods (such as [Boo94], [Jac92], [Rum93]) includes delegation explicitly as a concept of its own. We presume that this is mainly for two reasons: Overestimation of the expressive power provided by inheritance, and the fact that most object-oriented programming languages do not allow for a convenient and safe implementation of delegation. With this report we will focus on both aspects. We will first analyse the conceptual shortcomings of both inheritance and common associations. Then we will introduce delegation as a concept for object-oriented modelling. Furthermore we will demonstrate how to enhance Smalltalk in order to support delegation on the implementation level as well. On our experience delegation is a rather valuable modelling concept that may help to make a model more comprehensive and that fosters a system's flexibility and maintainability at the same time. However, in order to use it in an adequate way, it is necessary to consider its pitfalls as well.

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 the "is a" relationship also allows for 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. Single inheritance does not allow to express those semantic relationships. But what about multiple inheritance? Redesigning our small example using multiple inheritance would result in the generalization hierarchy presented in fig. 1.

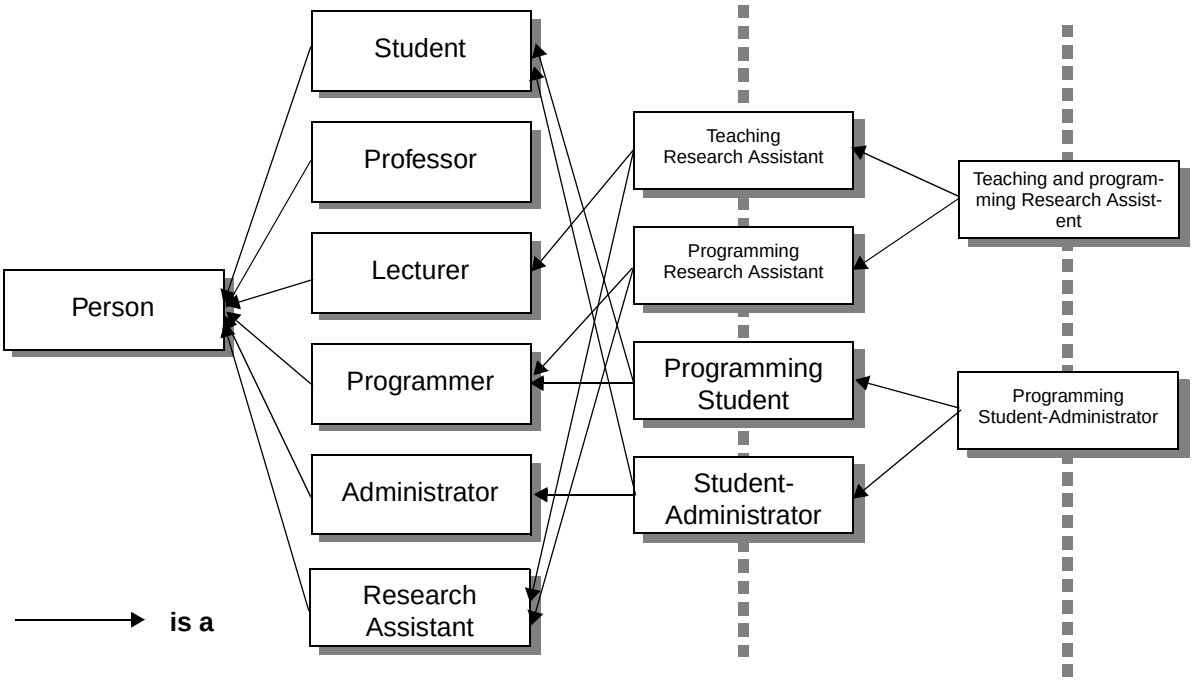


Fig. 1: Concepts resulting from multiple inheritance

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 modelling 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. It is hardly acceptable to add this sort of redundancy.

Our small example shows that 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" and the implementation of inheritance in common object-oriented programming languages. Natural language often does not explicitly differentiate between a concept and its instances: We say "a cat is a predator", no matter if we talk about a specific animal or the genus. 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 the class Cat is (usually) not an instance of the superclass Predator.

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.

3. Alternative Modelling Concepts

It is surprising that the problem we have discussed so far is hard to find in publications on object-oriented modelling. 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 instance - an instance of the class Bidder uses an instance of the class Person. Fig. 3 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 certain associations in the real world. 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.

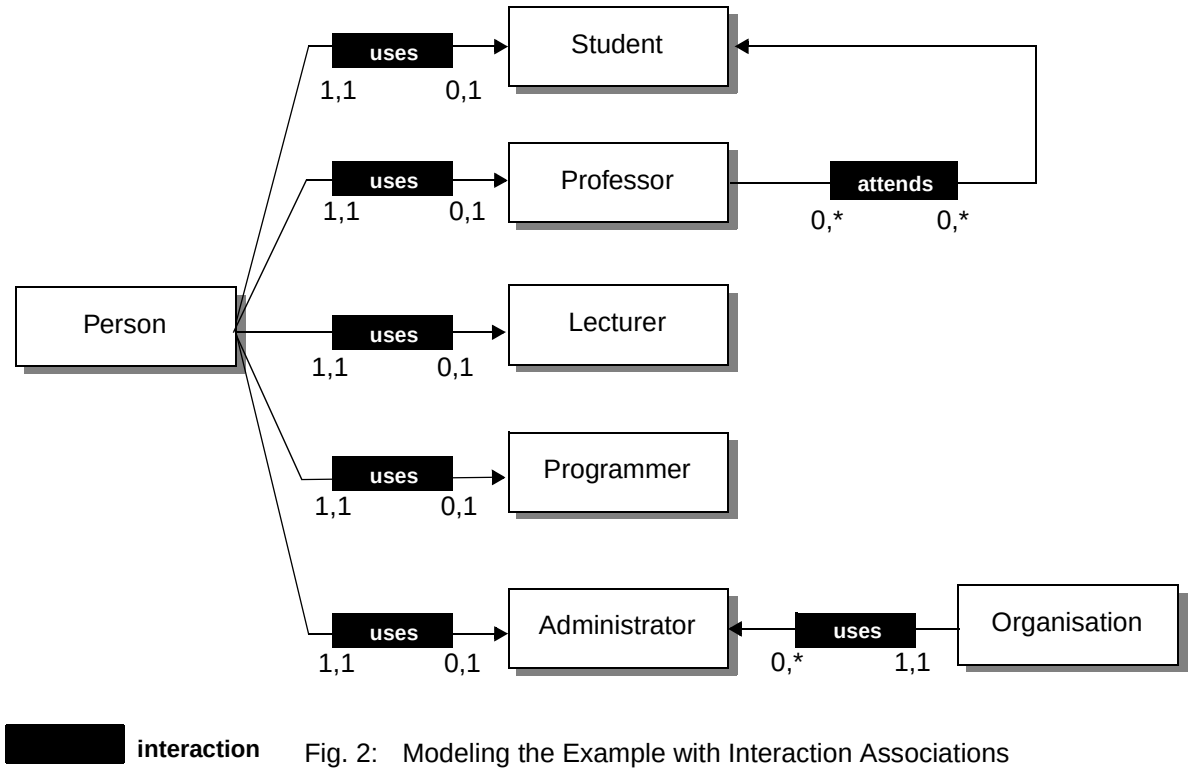


Fig. 2: Modeling the Example with Interaction Associations

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 (see fig. 4). Firesmith et al. only briefly mention 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.

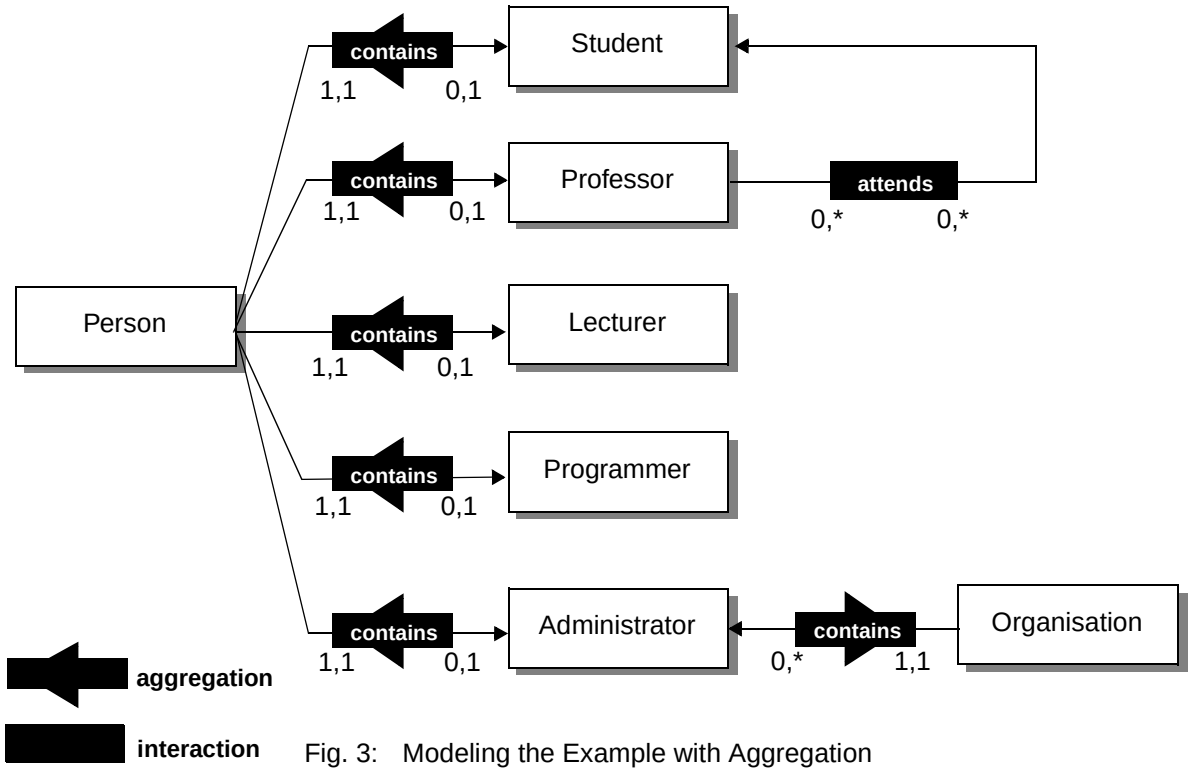


Fig. 3: Modeling the Example with Aggregation

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. Furthermore the features 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 roles which would be objects that were part of an instance of the class Person. Those objects would not need to include any states that are managed by instances of the class Person. Hence, redundancy could be avoided. However, aggregation 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 methodologies for object-oriented design (like [Rum93], [Boo94], [Jaco92], [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 any 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. So similar to interaction aggregation would force us to ab-

stract 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 denotations 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 than 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

Those authors who discuss delegation usually do not provide a precise definition. There are two different perspectives on the subject 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 instead of inheritance. 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 by "classless objects with delegation" ([Weg87]). Different from typical, class-based object-oriented languages classless languages - like Self - do not include the concept of a class. Instead every single object is being programmed directly. To take advantage of structural similarities an object can be created as a copy of an existing one. Afterwards both its state and behaviour may be changed. Classless programming languages are sometimes called delegation based, while objects within these languages are called prototypical objects ([Lie86], [Smi95]). 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. Goldberg and Rubin provide a typical characterization for this point of view: "When one object sends a message to a second object to fulfil one of its responsibilities. Delegation is an alternative to inheritance for sharing the behaviour of objects." ([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 depending on the context. Not only that implementation and conceptual level are usually not clearly differentiated. Furthermore there are alternative terms: Sciore uses "object specialisation" [Scio89] in order to express that a "specialized" object "inherits" behaviour 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" [SmUn95]. Kappel und 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").

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". In order to avoid the ambiguity that might be

caused by the fact that the term delegation is sometimes used in the opposite direction (see fig. 4) we considered to speak of "actor" and "role" instead. However, we were not satisfied with "actor": Not only that it is used in various different ways, furthermore it does not exactly match what a delegator in our context is meant to be. A delegator does not have to be an "acting" object. This is different with "role". While it is also used in object-oriented modelling with different semantics (a role usually serves as an annotation that characterizes the function of an object participating in an association), it is very well suited to describe the function of a delegate in a delegation association. Therefore we decided to stay with "role" and to use "role filler" instead of "actor".

We define delegation as a special association with the following general characteristics:

1. It 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 not only includes the 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. This is very much like in real life: You would hardly ask a programmer for the person he is assigned to in order to then ask this person for his name. Instead you would directly ask the programmer for his name. In case a role filler object includes a service that is already included in a role object's native interface 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 and a role class are by default inherited to their respective subclasses.

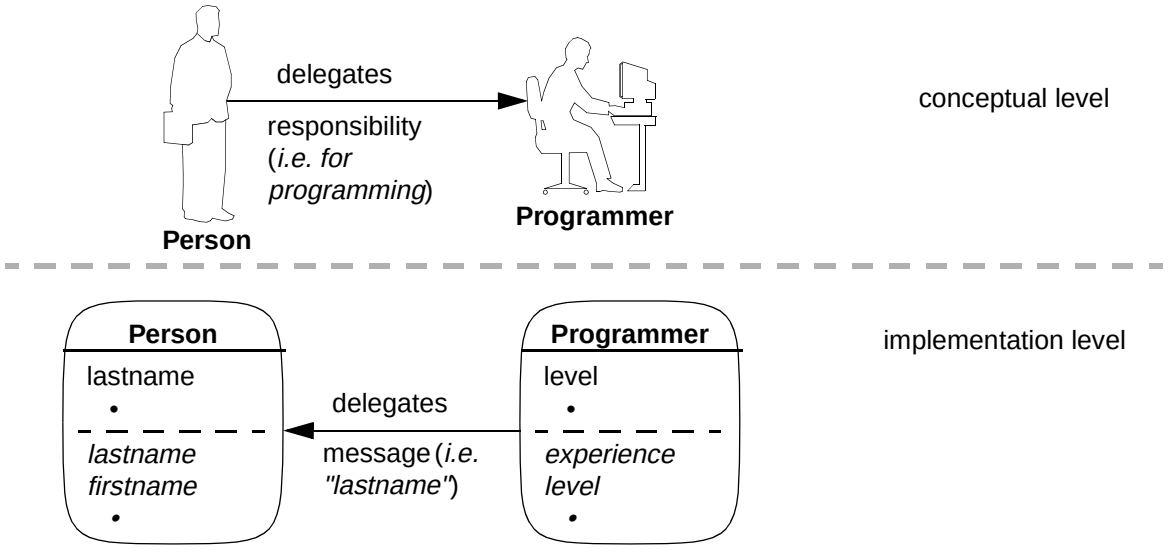


Fig. 4: Context dependent meaning of delegation

Furthermore 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. Furthermore the special semantics of both classes will require certain extensions on the implementation level (see below).

#2 A role filler may in general have none or many roles. Within a particular delegation the cardinality 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 modelling this situation). For another example see fig. 11.

#3 At a point in time a role must not be associated with more than one role filler. While it is obvious why a role should not be associated with more than one role filler of a particular class, one may find real world situations where it could be helpful to have one role associated with more than one role filler - of different classes - at the same time. For instance: If you need a class that represents students who are employed in a research project you might consider its instances as roles both of Employee- and Student-role fillers. We could speak of "multiple" delegation in this case. Nevertheless there are two reasons why we decided not to allow for it. First we suppose that "multiple" delegation usually will lead to vague and thereby misleading concepts. Furthermore additional strategies would be required to deal with naming conflicts.

To further specify the concept of delegation we will first discuss a number of possible constraints - which may be more or less adequate depending on the general attitude towards software engineering requirements. Similar to other modelling concepts (such as inheritance) there is a trade-off between flexibility and integrity. There are two main criteria that illustrate this conflict:

a) Number of role filler classes corresponding to one role class

- Pro integrity: Instances of a particular role class may be assigned only to an instance of *one* corresponding role filler class (and its subclasses respectively). This constraint fosters system integrity by allowing for statically checking an object model's consistency. In case a role class had more than one corresponding role filler class, it would hardly be possible to check whether a particular service is available during run-time: May be, the role object is associated with the required role filler object when it is needed, may be not ... Furthermore this constraint helps to protect people who deal with an object model against confusion. Note that it does not exclude to move a role during its lifetime from one role filler to another.
- Pro flexibility: Instances of a particular role class may be assigned to instances of more than one corresponding role filler class, but only to one instance at a time. This is rather common in the real world: Sometimes a role or a functionality respectively can be fulfilled by different role fillers (like a person or an institution).

b) Multi-level delegation

- Pro integrity: A role object must not act as a role filler object. If a role could be a role filler at the same time a model would get much more complex: A multi level delegation hierarchy would interfere with a multi level generalisation hierarchy. Such a model would be difficult to understand and hereby more difficult to maintain. However, with the first pro integrity constraint in place it would be possible to statically check whether a particular service was available with a role object during run-time - no matter whether the second constraint is valid or not.
- Pro flexibility: A role object may act as a role filler object, too. There are real world domains that suggest the use of multi-level delegation. For instance: A dean can be regarded as a role of a professor who in turn is a role of a person.

The decision for a specific definition of delegation certainly depends on individual preferences. We suggest the following compromise - resulting in two further constraints:

#4 The number of corresponding role filler classes is restricted to one.

If you allow instances of a particular role class to be assigned to instances of more than one corresponding role filler class the whole idea of transparently accessing a role filler's protocol through a role is jeopardized: Transparent access does not help much as long as you do not know which services are available. On the other hand there are relevant situations where a role can be assigned to role fillers of different classes. For instance: A customer may be regarded as a role of both a company or a person. The concept of delegation we have decided for does not allow for multiple role filler classes. That does not necessarily exclude to have a role associated with instances of different role filler classes - provided they are all subclasses of one common superclass. It may be helpful to define an abstract superclass for this purpose, thereby providing a minimum common protocol for all possible role fillers (see example 4.2 e) below).

#5 Multi-level delegation is permitted. However, cyclic associations are not permitted. In other words: By no means may a role object act as a role filler of itself.

In case the number of a role's corresponding role filler classes is restricted to one it seems appropriate to allow a role object to act as a role filler object: It may increase a model's complexity but is no serious threat to integrity (see above). For this reason it is not excluded by our definition of delegation.

Since we are focusing a conceptual rather than an implementation level so far we can neglect the question how the references from a role object to its role filler objects and vice versa are managed. The concept of delegation we propose can be summarized as follows:

Basic properties:

Delegation is a binary association with a role object and a role filler object.

The role object behaves like its role filler object by providing the role filler's interface and allowing for transparent access to the role filler's properties.

Constraints:

#1 Only classes that are kind of a special role class can be used to serve as roles within a delegation association.

#2 A role filler may in general have none or many roles of one class. The default is none or one.

#3 At a point in time a role must not be associated with more than one role filler.

#4 The number of corresponding role filler classes is restricted to one.

#5 Multi-level delegation is permitted.

In order to support modelling with delegation we enhanced our own object-oriented modelling method called MEMO ("Multi Perspective Enterprise Modelling", Frank 1994) with a special notation for expressing delegation (see fig. 5). Note that there usually is no need to explicitly assign cardinalities. There has to be exactly one role filler object (which is an instance of Person in our example) by definition. The default for associated role objects is min: 0, max: 1. (s. constraint #2). Only if the default is not acceptable you would explicitly assign a deviant cardinality. Each delegation association is assigned a name. The connecting arc defines the direction in which to read the association. In other words: If you regard the association as a predicate the arc starts at the subject and points to the object. The delegation symbol on the other hand serves to identify the role and the role filler: The role object is the one that is next to the half circle on top of the rectangle.

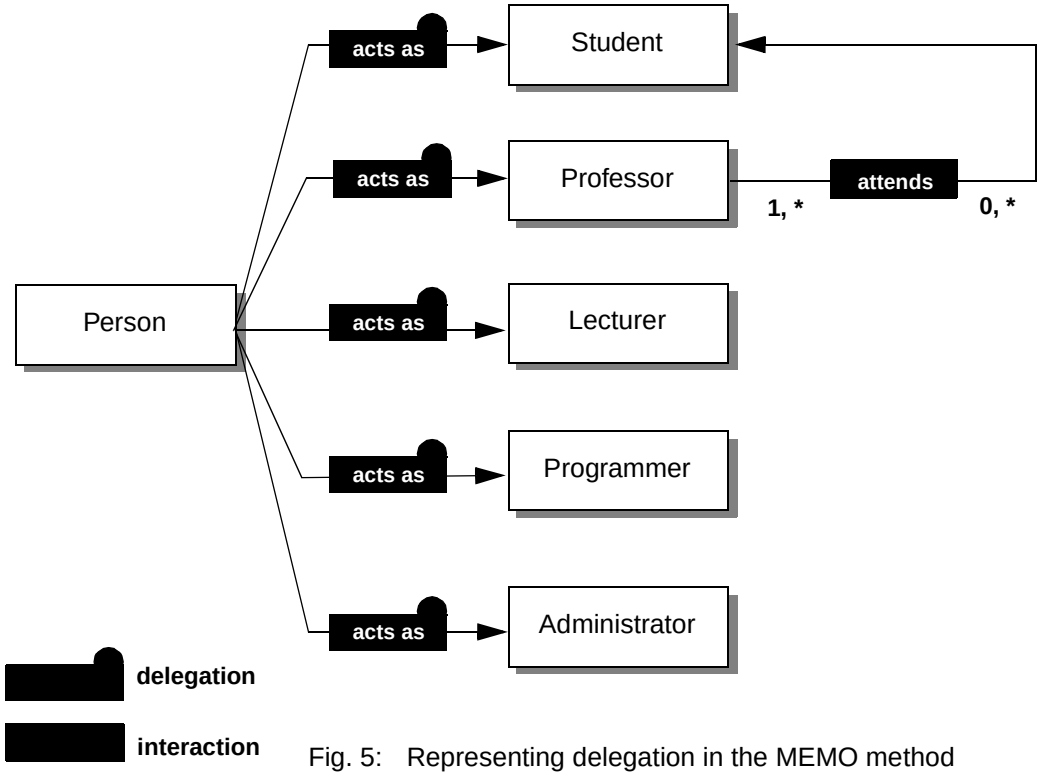


Fig. 5: Representing delegation in the MEMO method

4.2 Some Examples

The following examples serve to illustrate typical cases for delegation. At the same time some of them point to particular problems that may be accompanied by 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.

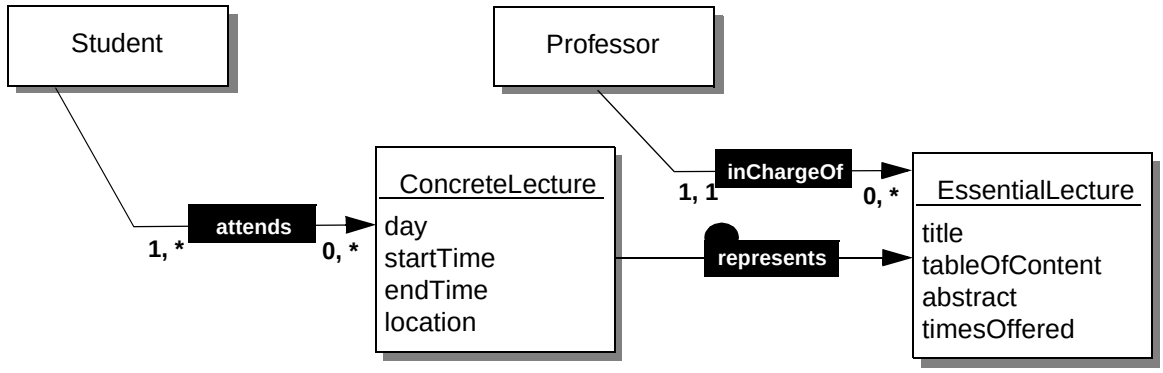
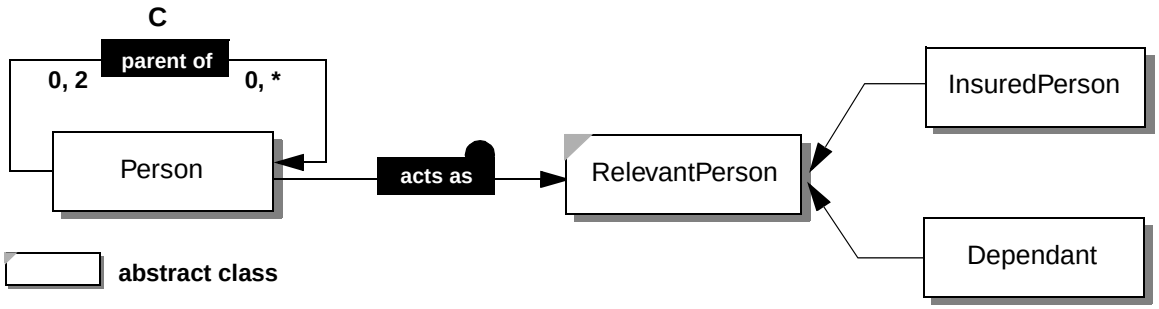


Fig. 6: Essential and concrete lectures

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 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 anyway 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. 7). In this case we would simply add a new role by creating an instance of InsuredPerson. The default cardinality for roles is 0, 1. Since it is not overwritten in our example the instance of Dependant would now have to be deleted. This would not affect relationships between cus-

tomers as long as those are modelled as associations between Person objects.



C An instance must not be associated with itself.

Fig. 7: 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 #4. On the other hand it may turn out that introducing two different kinds of customers without a common superclass will add redundancy, since there are 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: 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 default cardinality of roles again prevents a Customer object being associated with a Company object and a Person object at the same time. However, this example should make clear 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.

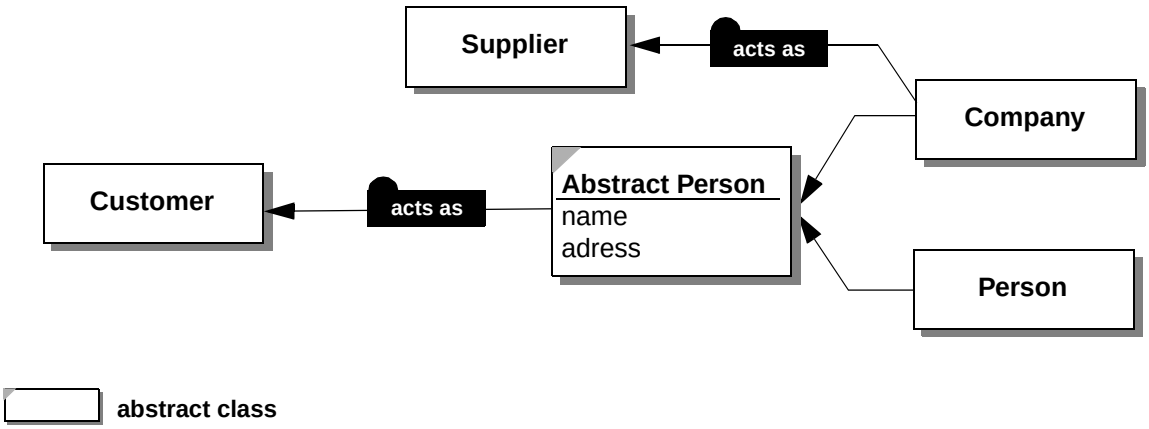


Fig. 8: "Multiple" Delegation through generalisation

d) Naming issues

Within an address management application you want to be able to store both a person's home address and his business address. The business address could be linked to or included in an instance of the class Employee while the home address could be directly linked to an instance of class Person. Fig. 9 shows a simplified example where we only consider a person's telephone numbers. There are two options for naming the services providing access to the home and the office phone numbers respectively. If there is no need to give access to an employee's home phone number through an Employee object (or you even want to avoid it), it is a good idea to use the same name for both services. In this case a message like "phoneNumber" sent to an Employee object would only return the phone number(s) assigned to the employee. If it is important to deliver both, an employee's office phone number and his home phone number, one should use appropriate names - like "officePhone".

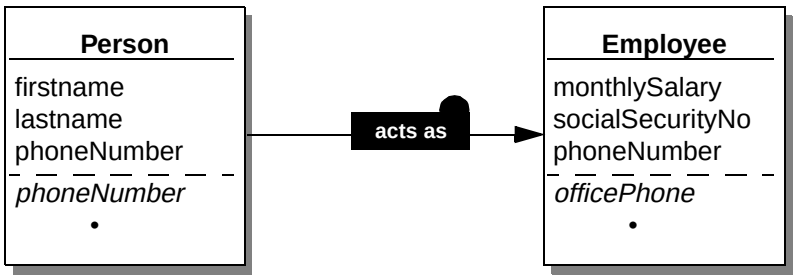


Fig. 9: Delegation and naming conventions

e) 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 professor 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 losing

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. 10). Note, however, that this compromise would lack a common abstraction from 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 constraint #4). At the same time Professor could not be a subclass of Employee - as long as you do not allow for multiple inheritance.

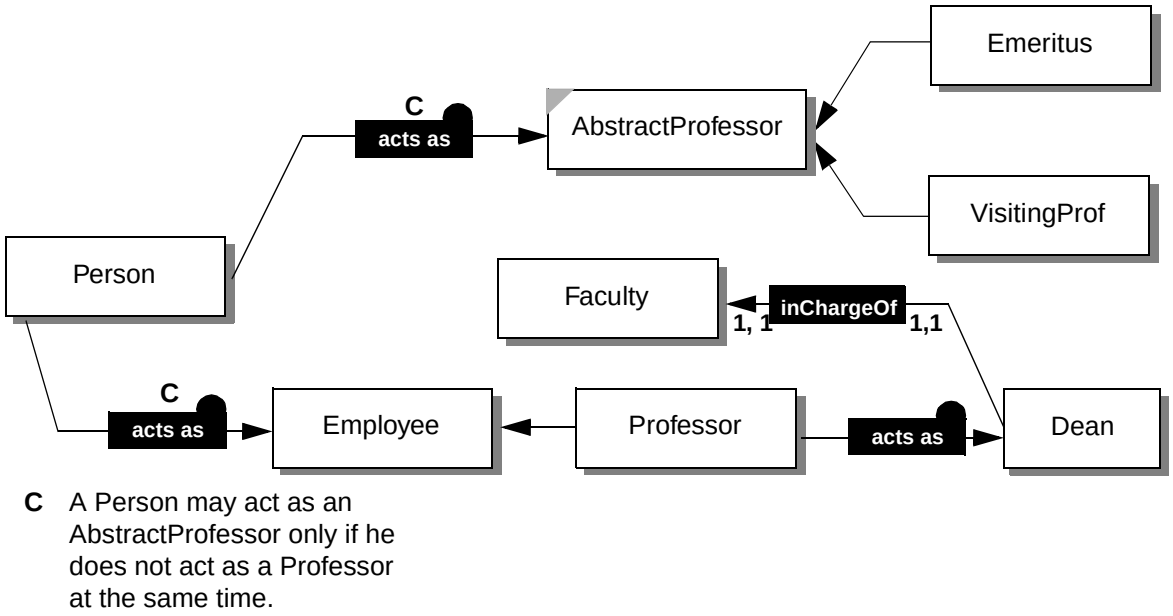


Fig. 10: Combining multi level delegation and inheritance

f) Role abstraction

If you look at example d) it may be relevant for a Person object to deliver all telephone numbers the particular person can be reached through - no matter which role they are assigned to. In other words: There should be a service like "allPhoneNumbers" that delivers all these numbers. On the implementation level this will require to make sure, that a Person object holds references to all its role objects. The implementation of delegation we will introduce below will provide these references by default. On the conceptual level you have to take into account that in this case you need to abstract from the fact that information, that might be associated with a person, is actually stored in role objects. Delegation provides an abstraction in the opposite direction. The only chance to foster role abstraction is by introducing naming conventions. First you would check, which of the role filler's services might be used with abstractions in mind that require to collect the corresponding information from all its roles. These services could then be implemented in all associated role classes - for instance a service "phoneNumber". In case it is appropriate this could be accomplished by defining a common (abstract) superclass. In order to allow for transparently accessing a Person object's phone numbers it would be necessary to implement additional access services with different names (for instance "homephone", see fig. 11).

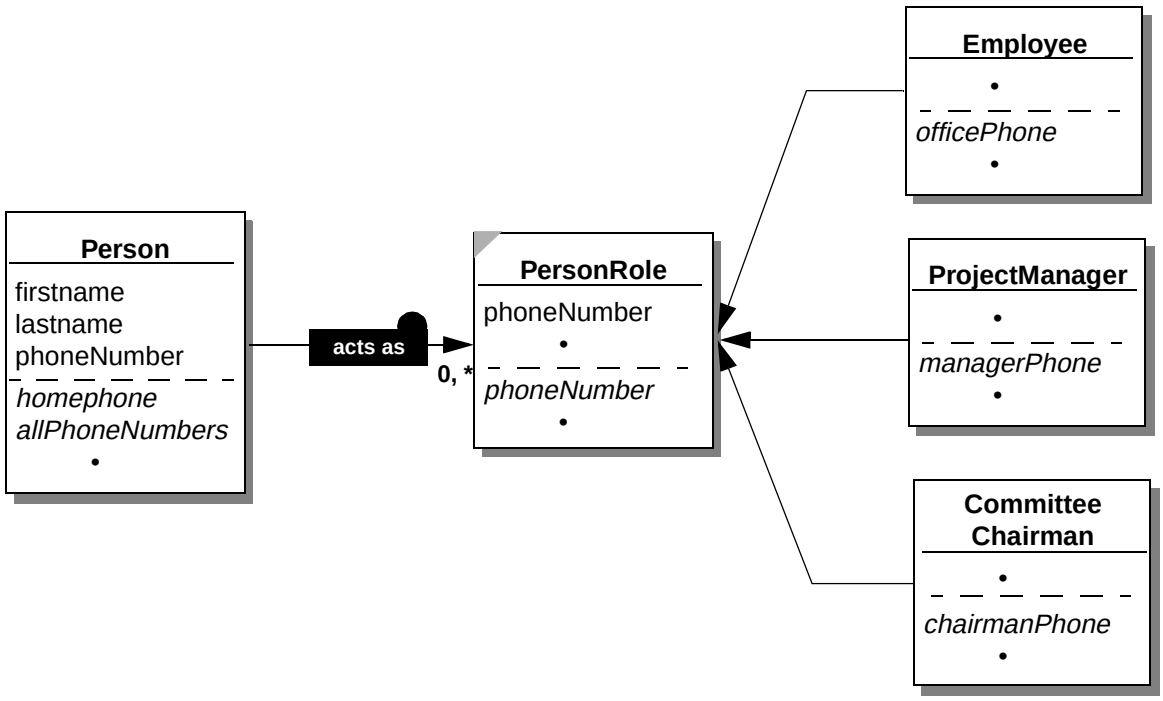


Fig. 11: Providing for role abstraction using an abstract role superclass

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 between inheritance and delegation Rumbaugh et al. suggest to focus 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) We find this criterion rather confusing. While one could argue whether it is a necessary condition for inheritance (see 5), it is certainly not relevant for differentiating between inheritance and delegation. Applying Rumbaugh et al.'s suggestion to our first example would for instance result in specifying the class Student as a subclass of the class Person - which is 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. 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 words 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. (In this case however, the lack of multiple delegation would imply to model Professor as a role of Person - thereby loosing the semantics that it might be a role of Employee as well. See 4.2, e)
- 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.

There are roles which may exclude each other in certain domains. For instance: It may be that a person who acts as a student cannot act as a professor in the same domain. Such potential conflicts can be detected during analysis simply by generating all possible combinations and checking each one for plausibility. In case you detect a conflict an appropriate constraint should be added to the model. 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: "Delegation is a good design choice only when it simplifies more than it complicates." ([Gam95], p. 21).

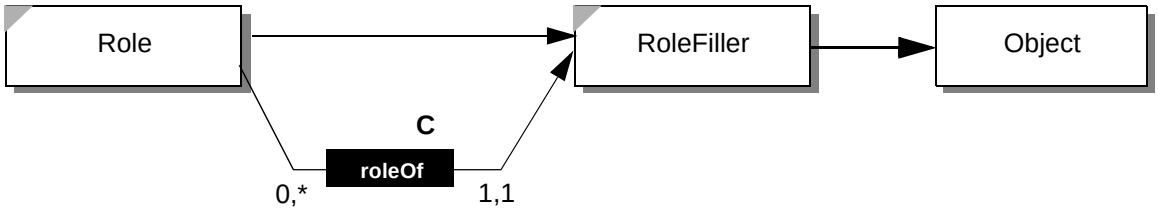
4.4 Enhancing Smalltalk with Delegation

Delegation is not only a valuable modelling concept from an academic point of view. Kathuria/ Subramaniam, who suggest a similar concept that they call "assimilation", state: "As practitioners we have a strong need for a concept like assimilation." ([KaSu96], p. 39). The conceptual advantages offered by delegation recommend to use it even if you do not have a programming language that supports it. Nevertheless, it is without doubt more attractive to use, if there is no semantic gap between object model and code. However, among the more popular object-oriented programming languages only Self features a concept that is similar to delegation ([SmUn95]) - *instead* of inheritance (see 4). In order to use delegation with programming languages which feature inheritance, it would be necessary to extend them with a corresponding mechanism. In the following sections we will demonstrate how such an extension can be accomplished. The implementation we introduce was done in VisualWorks©-Smalltalk. However, it can be easily adapted to other Smalltalk dialects.

4.4.1 Design

Constraint #1 (see 4.1) suggests not to allow arbitrary objects to serve as a role or as a role filler. In order to enforce this constraint we introduce two abstract classes, Role and RoleFiller. Any domain level role or role filler has to be explicitly defined as a subclass of one of those

abstract classes. Since we allow for multi level delegation (constraint #5), a role may act as a role filler, too. This aspect can be expressed by defining Role as a subclass of RoleFiller - since RoleFiller does not include any features that would corrupt a role's behaviour (see fig. 11).



C Cyclic associations have to be avoided.

Fig. 11: Preliminary object model for a delegation mechanism

In order to describe the behaviour of these classes in more detail, we will consider a number of design/implementation issues.

Transparent Message Dispatch

The essential behaviour to be implemented for roles is a transparent dispatch of certain messages to its role filler: Whenever a role object receives a message which is not implemented by a corresponding method neither in the role's class nor in any of its superclasses, the message should be forwarded to the role filler. With statically typed languages such as Eiffel it is necessary to define the messages to be dispatched at compile time. This could be accomplished by copying the interface of the role filler to its roles and by explicitly coding the dispatch for each message. This approach, however, is hardly satisfactory. Not only that code will be more difficult to understand (role objects will contain dispatch-messages that actually do not belong to their own responsibilities), furthermore software maintenance can be expected to be extremely cumbersome (whenever the interface of a role is modified, the interfaces of all of its roles have to be modified, too) - thereby endangering a system's integrity to an unacceptable extent.

Languages that feature dynamic typing, such as Smalltalk or Objective-C, are in principle better suited to be extended with delegation. The question that has to be dealt with at first place is, how to provide for a mechanism that allows for transparently dispatching messages from a role to its role filler. If a message is sent to a role object that does not contain this message in its protocol (which includes all the services inherited from its superclasses), this would normally cause an exception to be raised. Instead delegation would require to dispatch such a message to the associated role filler object. Hence it would be required to prevent the exception to be raised and to implement a message dispatch instead. Fortunately the Smalltalk code that raises the exception can be accessed and modified. In Smalltalk, an object that receives a message, it does not understand (in other words: the method lookup failed), will call its own method doesNotUnderstand: aMessage with the failed message (aMessage) as a parameter. The doesNotUnderstand:-method is implemented in `Object` (the superclass of all other classes in Smalltalk). The basic idea now is to replace the original implementation of this method with a method that dispatches the message to an associated role filler object (see fig. 12 and 4.4.2).

Referential Integrity

Within a delegation association both, a RoleFiller object, and a Role object should know each other. That suggests to store appropriate references to each other within a role filler and its roles. Implementing corresponding set methods, however, requires to prevent inconsistent states resulting from only partially set associations: Whenever a role filler adds a role to its set of roles, it should be made sure that the role in turn sets the role filler - et vice versa. This behaviour is implemented with the methods addRole: aRole in RoleFiller, and roleFiller: aRole in Role (see 4.4.2).

Referential integrity is a requirement for delete operations, too. The methods removeRole: aRole within RoleFiller, and roleFiller: aRoleFiller (where aRoleFiller would replace and thereby delete the previous role filler) within Role make sure, that deleting a reference will always include the deletion of its counterpart. Different from a Role object a RoleFiller object may hold many of such references (note that a role may act as a role filler as well). For this reason removing it (having it collected as garbage), requires to break all references to its roles. The release method within RoleFiller provides this behaviour. Since a role may act as a role filler as well, the release method implemented with Role first calls the release method implemented with its superclass, and then removes the reference to the associated role filler. Notice, however, that this method has to be enhanced within subclasses, if those subclasses allow to establish additional references (which will usually be the case). Handling delegation associations furthermore requires to enforce the constraint, that there must not be any cyclic associations. This purpose is served by the method roleFiller: aRoleFiller, implemented with Role (see 4.4.2).

Meta Information

One of Smalltalk's outstanding features is to provide meta information during runtime. Any object, for instance, provides a method that delivers its class or its superclass. The method isKindOf: aClass allows to find out, if an object is a subclass of a certain class aClass - thereby providing the behaviour (or, to be more precise: the interface) of this class. In order to check, if a Role object provides the interface of another class, isKindOf: aClass is certainly not satisfactory: A role provides transparent access not only to the interface of its superclasses but also to the interface of its role filler. This is similar with the method respondsTo: aMessage, that allows to find out, whether aMessage is included in an object's (or any of its superclasses') interface. Since both the semantics of isKindOf: aClass and respondsTo: aMessage are well known in the Smalltalk community, it is definitely no good idea, to replace them in Role with deviant implementations. Instead we added two methods to Role. behavesLike: aClass first calls isKindOf: aClass. If that call fails, it will send behavesLike: aClass to the associated RoleFiller object. repliesTo: aMessage, proceeds in the same way, calling respondsTo: aMessage instead (see 4.4.2). Checking the existence of a certain interface does not always imply to know whether an object is a role. In order to avoid an additional check by calling isKindOf: Role, the two methods added to Role could also be added to Object, where they would simply call isKindOf: aClass or respondsTo: aMessage. Notice, however, that changing basic classes always causes a price to be paid for configuration management.

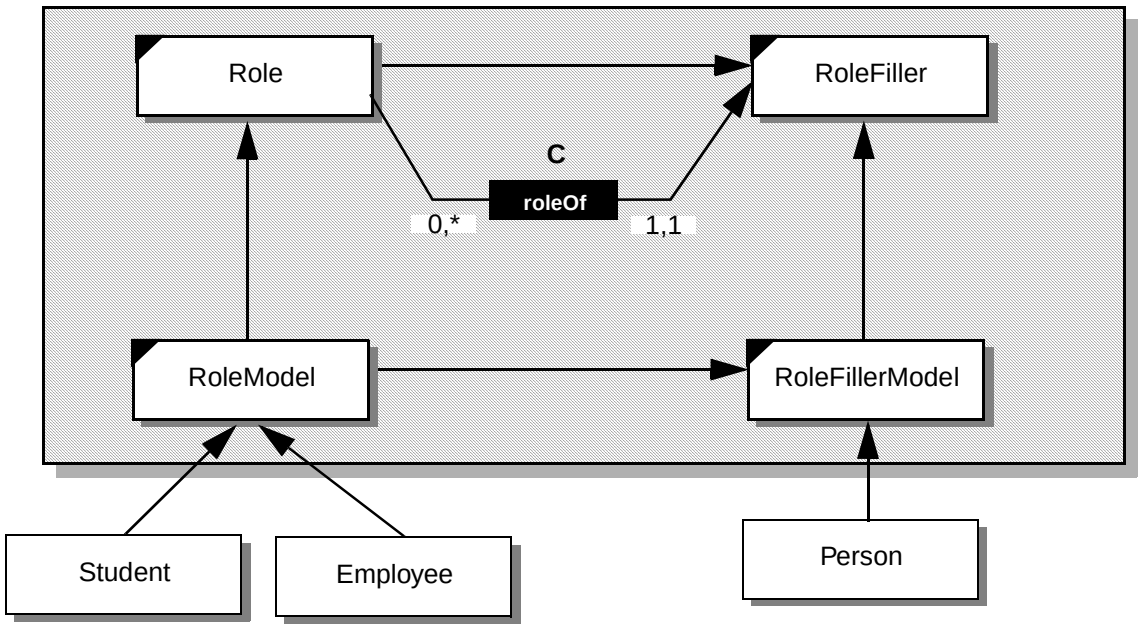
Dependence Mechanism

Similar to the previous section this is a Smalltalk specific consideration, too. The dependence mechanism allows to define an object as a dependant of another object. Whenever an object changes its state, its dependants will be informed transparently. For a detailed description of

the dependence mechanism see ([Parc95], pp. 237, [Howa95], pp. 16). Introducing the delegation concept adds further complexity to dependence associations. Consider, for example, an object that expresses its interest in changes of a Programmer object, which in turn is a role of a Person object. That object will only be notified if the Programmer object changes - although the conceptual level would suggest an interest in the associated role filler as well, since a role represents its role filler in a certain context. Often it would be desirable to have a Role object's dependants transparently notified about the associated RoleFiller object's (which may involve more than one object in the case of multi level delegation) changes as well. For this reason we specialized two additional classes, RoleFillerModel and RoleModel, from RoleFiller and Role respectively. These classes allow for transparent management (registering, 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 (see fig. 12) 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 specialise from RoleFillerModel and RoleModel, otherwise you would use the framework by defining subclasses from RoleFiller and Role.

The Framework

The framework presented in fig. 12 is intended to be used by the so called "complete" strategy. Different from the "use as is" strategy it requires the developer to add additional code - mainly by specializing certain classes within the framework. We do not encourage to use the framework in a "customise" fashion: The classes of the framework should not be modified - not only because they define the essential semantics of delegation, but also to prevent maintenance problems that may occur with upcoming releases of the framework. For a detailed discussion of basic strategies to use frameworks see [CoPo95].



C Cyclic associations have to be avoided.

Fig. 12: The delegation framework and an example of how to use it via specialization.

To make use of the framework you would simply specialise the concrete role filler from class RoleFiller and the roles from class Role, or - if you want to take advantage of the dependence mechanism - from class RoleFillerModel and RoleModel respectively. Going back to our example shown in figure 3 class Person could be defined as a subclass of class RoleFiller and the classes Professor, Student, Lecturer and Administrator could be subclasses of class Role (maybe with an additional abstraction like PersonRole, as a superclass of Professor, Student, etc.).

4.4.2 Implementation

The following section describes the essential code of the delegation framework. The description is restricted to the classes Role and RoleFiller, since RoleModel and RoleFillerModel only add behaviour that is well known from class Model. The complete implementation of the framework together with a comprehensive example can be obtained via http://www.uni-koblenz.de/~iwi/delegation.

Implementation of Class Role

The instance level protocols both of Role and of RoleFiller are shown in fig. 13. The methods corresponding to the services printed in italic are listed below.

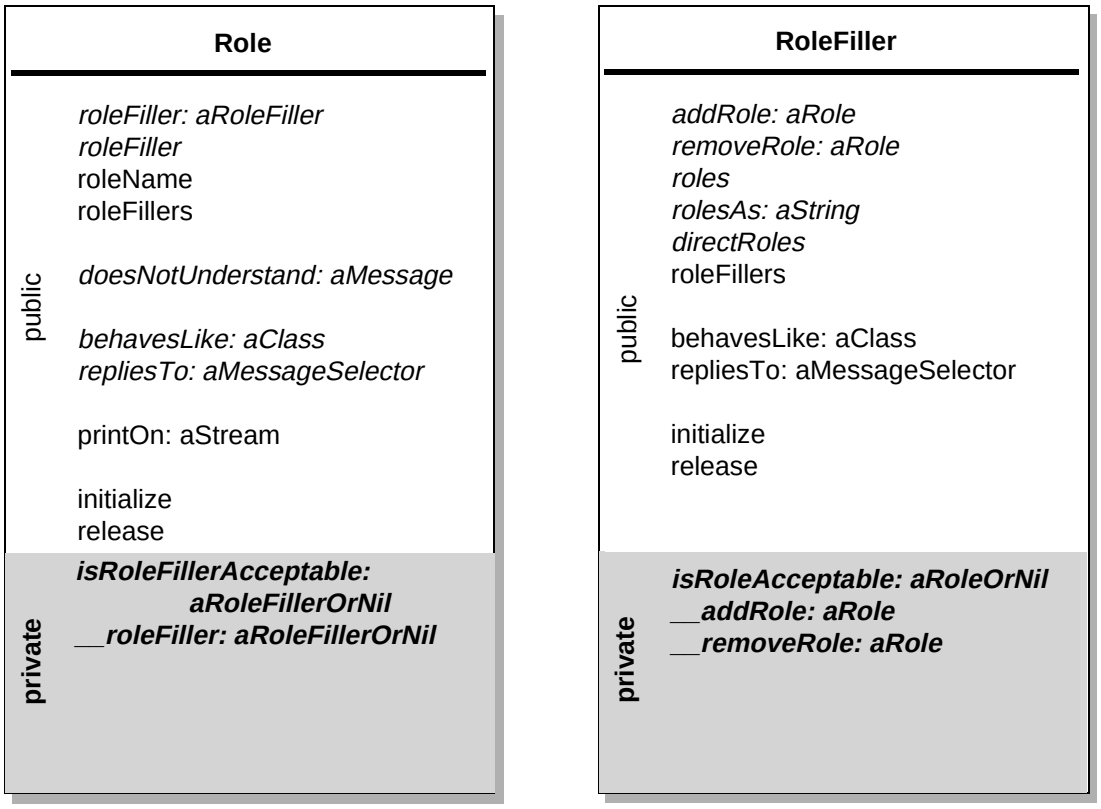


Fig. 12: Protocols of Role and RoleFiller

The class definition for RoleFiller:

```
Object subclass: #RoleFiller
    instanceVariableNames: 'roles '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'DelegationConcept'
```

The class definition for Role:

```
RoleFiller subclass: #Role
    instanceVariableNames: 'roleFiller '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'DelegationConcept'
```

The original implementation of doesNotUnderstand: aMessage in class Object:

```
doesNotUnderstand: aMessage

    "The default behaviour is to create a Notifier containing the
    appropriate message and to allow the user to open a Debugger.
    Subclasses can override this message in order to modify this behaviour."

    | selectorString |
    selectorString :=
        Object errorSignal
            handle: [:ex | ex returnWith: '** unprintable selector **']
            do: [aMessage selector printString].
    Object messageNotUnderstoodSignal
        raiseRequestWith: aMessage
        errorString: 'Message not understood: ' , selectorString.
    ^self perform: aMessage selector withArguments: aMessage arguments
```

The replacement of that method within Role:

```
doesNotUnderstand: aMessage

    "If the receiver does not understand the message aMessage propagate it to the receiver's
roleFiller."
    ">>> Types: aMessage : Message"
    ">>> Returns: Object"

    ^self roleFiller
            perform: aMessage selector
            withArguments: aMessage arguments
```

The methods to get and set the role filler within Role:

roleFiller: aRoleFillerOrNil

```
    "Set the receiver's role filler to aRoleFillerOrNil. If the role filler of the receiver is not nil
break the existing reference prior to establishing the new one."
    ">>> Types: aRoleFillerOrNil : RoleFiller | nil"
    ">>> Exceptions:
        Notification (The role filler <aRoleFillerOrNil> is not acceptable!)
        Notification (The role <self> is not acceptable!')"


    "If the receiver's current role filler is equal to aRoleFillerOrNil do nothing."
    self roleFiller == aRoleFillerOrNil ifTrue: [ ^self ].


    "Test whether or not the role filler is acceptable for the receiver and the receiver is accept-
able for the role filler."
    (self isRoleFillerAcceptable: aRoleFillerOrNil)
        ifFalse: [ self notify: 'The role filler ', aRoleFillerOrNil printString, ' is not acceptable!'.
^self ].
    (aRoleFillerOrNil isNil or: [aRoleFillerOrNil isRoleAcceptable: self])
        ifFalse: [ self notify: 'The role ', self printString, ' is not acceptable!'. ^self ].


    "Protected establish the reference from the receiver to the role filler and vice versa."
    self __roleFiller: aRoleFillerOrNil.
    (self roleFiller isNil) ifFalse: [ self roleFiller __addRole: self ].
```

roleFiller

```
    "Return the receiver's role filler."
    ">>> Returns: RoleFiller | nil"
    ^roleFiller
```

The implementation of the private methods for Role:

isRoleFillerAcceptable: aRoleFillerOrNil

```
    "Returns true if aRoleFillerOrNil can be accepted as the receiver's role filler, false other-
wise."
    ">>> Types: aRoleFillerOrNil : RoleFiller | nil"
    ">>> Returns: Boolean"


    "If the argument is nil it can be accepted."
    aRoleFillerOrNil isNil ifTrue: [ ^true ].
    "Don't allow any cycles."
    (aRoleFillerOrNil == self or: [(aRoleFillerOrNil roleFillers includes: self)])
        ifTrue: [ ^false ].
```

```
"Check if aRoleFillerOrNil is kind of RoleFiller"
(aRoleFillerOrNil isKindOf: RoleFiller ) ifFalse: [ ^false ].
^true
```

__roleFiller: aRoleFillerOrNil

```
"Assign aRoleFillerOrNil as the receiver's role filler in a quasi protected mode. If the receiver's role filler is not nil break the existing reference prior to establishing the new one."
">>> PROTECTED: RoleFiller"
">>> Types: aRoleFillerOrNil : RoleFiller | nil"

(self roleFiller isNil)
    ifFalse: [ self roleFiller __removeRole: self ].
roleFiller := aRoleFillerOrNil.
```

The method for requesting meta information for Role:

behavesLike: aClass

```
"Returns whether or not the receiver behaves like aClass, i.e. answers the question whether or not the receiver or the receiver's role filler is kind of aClass."
">>> Types: aClass : Class"
">>> Returns: Boolean"
^(self isKindOf: aClass) or: [self roleFiller behavesLike: aClass]
```

repliesTo: aMessageSelector

```
"Answer whether the method dictionary of the receiver's class or the method dictionary of the class of the receiver's role filler contains aMessageSelector as a message selector."
">>> Types: aMessageSelector : Symbol"
">>> Returns: Boolean"
^(self class canUnderstand: aMessageSelector)
        or: [ self roleFiller repliesTo: aMessageSelector ]
```

The methods to add, remove and retrieve roles within RoleFiller:

addRole: aRole

```
"Add aRole as one of the receiver's roles."
">>> Types: aRole : Role"
">>> Exceptions:
    Notification (The role  <aRole> is either nil or not a role!)
    Notification (The role filler <self> is not acceptable!)"
"Check if the association can be established from both sides."
(self isRoleAcceptable: aRole)
    ifFalse: [ self notify: 'The role ', aRole printString, ' is either nil or not a role!'. ^self ].
(aRole isRoleFillerAcceptable: self)
    ifFalse: [ self notify: 'The role filler ', self printString, ' is not acceptable!'. ^self ].
```

```
"Protected establish the association."
self __addRole: aRole.
aRole __roleFiller: self.
```

removeRole: aRole

```
"Remove aRole as one of the receiver's roles. If the role is removed the association from
aRole to the receiver is also broken, but aRole isn't released!!! (This must be done manually)"
>>> Types: aRole : Role"

(self __removeRole: aRole)
    ifTrue: [ aRole __roleFiller: nil]
```

directRoles

```
"Return all roles the receiver references directly through the instance variable roles."
">>> Returns: Set of Role"
^roles
```

roles

```
"Return a collection including all roles of the receiver. This includes also all roles of the re-
ceiver's roles."
">>> Returns: Set of Role"
| set |
set := Set new.
set addAll: self directRoles.
self directRoles do: [ :role | set addAll: role roles ].
```

rolesAs: aString

```
"Return all roles that have class names matching aString - aString can include any wild-
cards."
">>> Types: aString : String"
">>> Returns: Set of Role"
">>> Example: If there are role classes Professor and Programmer the message <role fill-
er> rolesAs: 'Pro*'  will return instances of both (if any)."
^self roles select: [ :role | (aString match: role class name asString) ]
```

The implementation of the private methods for RoleFiller:

isRoleAcceptable: aRoleOrNil

```
"Return whether aRoleOrNil can be accepted as one of the receiver's roles."
">>> Types: aRoleOrNil : Object"
">>> Returns: Boolean"
"nil is not allowed as role."
```

```
    aRoleOrNil isNil ifTrue: [ ^false ].
    "Check if aRoleOrNil is kind of Role."
    (aRoleOrNil isKindOf: Role ) ifFalse: [ ^false ].
```

```
__addRole: aRole
    "Add aRole as one of the receiver's roles in a quasi protected mode."
    ">>> PROTECTED: RoleFiller"
    ">>> Types: aRole : Role"
    ">>> Returns: Boolean"
    ^(self directRoles includes: aRole)
        ifTrue: [ false ]
        ifFalse: [ self directRoles add: aRole. true ].
__removeRole: aRole
    "Remove aRole as one of the receiver's roles in a quasi protected mode."
    ">>> PROTECTED: RoleFiller"
    ">>> Types: aRole : Role"
    ">>> Returns: Boolean"
    ^(self directRoles includes: aRole)
        ifTrue: [ self directRoles remove: aRole. true ]
        ifFalse: [ ^false ]
```

4.5 Delegation as a Design Pattern

Finally we will present delegation as a design pattern. This did not happen because describing delegation as a design pattern would provide relevant additional information. However, the idea of design patterns, which can hardly be regarded as a concept of its own, can be helpful for a number of reasons. First it provides a structure that helps to produce a systematic documentation of a particular concept. With design patterns gaining popularity an increasing number of people is familiar with this sort of documentation. For this reason a design pattern can help with understanding a concept and with comparing it to related concepts. The structure we use is adapted from the one suggested by Gamma et al. [Gam95].

Pattern Name

"Delegation"

Also Known As

"object specialization", "instance level inheritance", "propagation"

Purpose

Delegation serves two purposes: During system design it allows to express a specific relationship between objects, thereby enhancing an object model's semantics. By enriching a programming language with a corresponding concept the semantic gap between design and implementation can be avoided.

Motivation

Sometimes an object of a certain class may have associated *occurrences*, which are only relevant within specific contexts: For instance a person may act as an employee in one context, as a customer in another context. Those occurrences are not only expected to behave like a person, but also to provide transparent access to the person's characteristics: You normally would ask an employee for his name instead of addressing "his" person. This sort of relationship is very similar to the relationship between a role filler and his roles. Inheritance does not allow to model this relationship in an appropriate way (see 2). This is also the case for common associations on the instance level (see 3) Delegation is a concept that allows to model relationships between roles and role fillers in a natural way. It is a binary association with one object (the "role" or "role object") that provides transparent access to the state and behaviour of another object (the "role filler" or "role filler object"). Another important difference from inheritance is to be seen in the fact that roles can be assigned to role fillers during run-time. For details see 4.1.

Applicability

see 4.3

Structure

see 4.4

Participants

In principle any class can be specialized from RoleFiller, Role, RoleFillerModel, or RoleModel. However, in order to foster a domain model's integrity it is recommended to decide in time which classes are permitted to be role or role filler candidates (see 4.3).

Dynamics

Whenever a Role object receives a message that is not included in its own protocol, it will dispatch this message to its RoleFiller object. In case the RoleFiller object cannot answer this message an appropriate message (like "do not understand") should be returned to the Role object which will return it to the original caller.

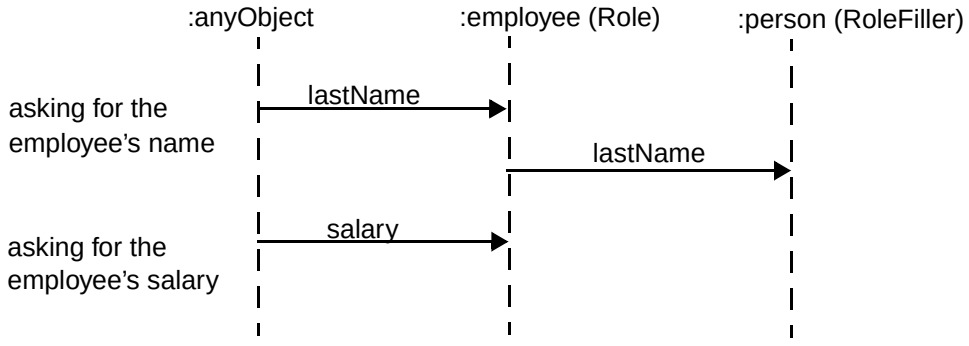


Fig. x: Example of an interaction between any object and a role object.

Consequences

An object-oriented programming language enhanced with delegation will allow a programmer to use delegation in about the same way as inheritance. Concrete role and role filler objects have to be defined through specialization. Then an association between the involved objects has to be established. After those declarations delegation can be used in a transparent way. The additional abstraction provided by the delegation mechanism requires a performance price to be paid. This is especially the case for multi level delegation.

Implementation

see 4.4

Sample Code

see 4.4

Known Uses

see 4.2

Related Patterns

The "client specified self" pattern, suggested by [Vil95], provides a mechanism that dispatches messages sent to one object to another object. This is accomplished by replacing the pseudo-variable "self" with a variable that can be set by the object that owns it. By assigning another object to this variable, the methods of this object can be used as if they were contained in the calling object's protocol. Compared to delegation this pattern is certainly less flexible, since it will only "transparently" dispatch messages which are explicitly sent to the "client specified self". Beck describes a pattern that is similar to delegation in some respect, but certainly does not serve the same purpose. It is called "instance specific methods" and allows to dynamically change the methods of an instance [Bec92]. From a conceptual point of view such an approach is certainly questionable. Another pattern, that is somehow related to delegation is "class migration" or "dynamic classes" [Wie95]. However, it does not help with most of the requirements fulfilled by delegation. Furthermore the idea of an object changing its class seems to be rather confusing.

5. Concluding Remarks

Delegation is an important concept to enrich both conceptual models and languages used on the implementation level. While delegation - although not always in a unique way - has been subject of many publications, popular object oriented modelling methods (like [Boo94], [Jac92], [Rum93]) do not include it as a concepts of its own. This is also true for recent efforts to suggest "unified" or "open" (and eventually standardized) modelling languages (like [FiHe96], [Rat97]).

One essential motivation to introduce delegation is to be seen in the shortcomings of inheritance to model certain aspects of the real world. However, inheritance is not specified in a unique way - if it is specified at all. The concept of inheritance we used in this report is adapted from common object-oriented programming languages such as Smalltalk, Eiffel, C++, etc. With this type of inheritance, sometimes called "intentional inheritance", the concept of a

class, its conceptual description, is inherited to its subclasses. An instance of a subclass is not an instance of the superclass. This is different with 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. According to specific object characteristics this set can be divided into subsets, subsets of subsets, etc. Each subset forms a subclass. Different from intentional inheritance each instance of a class is an instance of the corresponding superclass at the same time. Kappel and Schrefl describe the different semantics used with inheritance in a comprehensive way ([KaSc96], pp. 15).

It is important to note that extensional inheritance provides features that are very similar to delegation. For instance: Both Employee and Student could inherit from Person, but there instances would be instances of Person at the same time. An Employee object of this kind would behave in the same way as a role. Extensional inheritance is featured by database systems that extend the capabilities of RDBMS in an object-oriented way - like so called "Object-Relational" DBMS ([StMo95], [Cha96]). With the latest version of SQL ([Mat96]) supporting extensional inheritance as well, it may well be assumed that the next generation of mainstream data base systems will make extensional inheritance a central concept for managing persistent objects. At first sight such a perspective may suggest that delegation will become obsolete. However, it might even promote the future importance of delegation. Object-oriented programming languages do not support extensional inheritance. Furthermore 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, it could be an interesting option to map extensional inheritance to delegation et vice versa.

Acknowledgements

We would like to thank Michael Prasse for his valuable, and stimulating comments.

References

[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. 2. Aufl., Redwood City: Benjamin Cummings 1994

[Bec92] Beck, K.: Instance-Specific Behavior: How and Why. In: Smalltalk Report. Vol. 2., No. 6, 1992

[Cha96] Chamberlin, D.: Using the New DB2: IBMs Object-Relational Database System. San Francisco: Morgan Kaufmann 1996

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

[FiHe96] Donald Firesmith, D.; Henderson-Sellers, B.; Graham, I.; Page-Jones, M.: OPEN Modeling Language (OML) - Reference Manual. Version 1.0. MS-Word- or Postscript-Document, obtained via http://www.csse.swin.edu.au/cotar/OPEN/OPEN.html, Dec. 8th, 1996

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

[Gam95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Reading/Mass. etc.: Addison-Wesley 1995

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

[Howa95] 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

[Mat96] Mattos, N.M.: An Overview of the SQL3 Standard. Database Technology Institute. IBM - Santa Teresa Lab., San Jose/Ca. 1996

[Parc95] ParcPlace-Digitalk: VisualWorks User's Guide, 1995

[Rat97] Rational: Unified Modeling Language. UML Semantics. PDF-Document, obtained via http://www.rational.com/ot/uml/1.0/index.html, Jan. 1997

[Rum93] Rumbaugh, J. et al.: Object Oriented Modeling and Design. Englewood Cliffs/NJ: Prentice Hall 1993

[Scio89] Sciore E.: Object specialization. In: ACM Transactions on Office Information Systems, Vol. 7, No. 2, April 1989, pp. 103-122

[Smi95] Smith, W.A.: Using a Prototype-based Language for User Interface: The Newton Projects Experience. In: Proceedings of the OOPSLA95. New York: ACM 1995, pp. 61-72

[SmUn95] Smith, R.B.; Ungar, D.: Programming as an Experience. The Inspiration for Self. In: Proceedings of the ECOOP 95

[StMo95] Stonebraker, M.; Moore, D.: Object-Relational DBMSs: The Next Great Wave. San Francisco: Morgan Kaufmann 1995

[Ver92] Veryard, R.: Information Modelling. Practical Guidance. New York, London etc.: Prentice Hall 1992

[Vil95] Viljamaa, P.: Client-Specified Self. In: Coplien, J.O.; Schmidt, D.C.: Pattern Languages for Program Design. Reading/Mass. et al.: Addison-Wesley 1995, pp. 495-504

[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