**Friedrich Steimann**

# Role = Interface: A Merger of Concepts

**Interfaces are a prominent OO programming concept, since they allow the decoupling of specification and implementation. Roles, on the other hand, are a popular OO modeling concept. For example, UML has rolenames, classifier roles, association roles, and association end roles. Although roles and interfaces appear unrelated at first glance, it is shown that they have much in common—in fact, with a few changes in definition, the two concepts can be merged into one.**

It has been noted over and over that certain classes found in OO designs are not really classes, but roles. Typical examples are Customer, Employee, Passenger, etc. (roles of persons) and Product, Commodity, Asset, etc. (roles of things). Roles, unlike classes, have something dynamic about them; for example, through the course of its lifetime, an instance of class Person may successively become a student, an employee, and a retiree. Also, unlike with classes, belonging to (or playing, to stick with the role metaphor) several roles simultaneously is not unusual for the instances of certain classes: a person may simultaneously be a customer, a supplier, an employee, and a stockholder, perhaps a parent, and many other things too.

It has been criticized that mainstream OO programming languages lack the possibility of dynamic and multiple classification. However, this is not quite true. In fact, it is untrue, depending on how classification is defined. If it is defined as an object being an instance of a class, then dynamic classification (i. e., the change of classes after the instance has been created) is not possible in statically typed programming languages like C++ and Java, and multiple classification is limited to the object's being an instance of a certain class and all its superclasses. If, on the other hand, classification of an object is defined as that object belonging to the dynamic extent of a certain type, i.e., as being assigned to a variable of that type, then classification is both multiple and dynamic: multiple, because in languages such as Java instances of a class can not only be assigned to variables declared of this class and its superclasses, but also to variables declared of the interfaces the class implements; and dynamic, because the object joins and leaves the dynamic extents of types (the "classes," in the sense of classification) by means of assignment and substitution. The only restriction on dynamicity is that it must be statically enabled by the type (class and interface) declarations of a program. Such a restriction makes sense since dynamic classification is subject to certain natural, static conditions. For example, a piece of equipment should not be classified as an animate, no matter how dynamic dynamic classification is.

The Unified Modeling Language (UML) does not commit itself to either static or dynamic nor single or multiple classification. Nowhere in the UML specification does this become as clear as in the context of collaborations: a classifier role is a classifier like a class or interface, but "since the only requirement on conforming instances is that they must offer operations according to the classifier role, as well as support attribute links corresponding to the attributes specified by the classifier role, and links corresponding to the association roles connected to the classifier role, they may be instances of any classifier meeting this requirement."[1] In other words, a classifier role allows any object to fill its place in a collaboration no matter what class it is an instance of, if only this object conforms to what is required by the role. Classification by a classifier role is multiple since it does not depend on the (static) class of the instance classified, and dynamic (or transient) in the sense above: it takes place only when an instance assumes a role in a collaboration.[2]

This suggests that roles, like classes, classify objects, but unlike with class-based classification, role classification is multiple and, depending on definition, dynamic. It also suggests that roles are only partial specifications of the objects playing them. These objects can usually play several roles and accordingly have many partial specifications that all add to the total specification of the classes they are an instance of. Finally, it suggests that a role can be played by instances of different classes that are not related by inheritance and, in particular, that role playing is independent of implementation. Now if all this is really the case, then the properties of roles are the very properties of interfaces (interfaces as types, in the sense of Java and UML), and roles and interfaces are largely the same concept.

## ROLES AND INTERFACES IN OOP

In the introduction of their seminal book on design patterns, Gamma put forward the "program to an interface, not an implementation" principle that they paraphrase as follows: "Don't declare variables to be instances of particular concrete classes. Instead, commit only to an interface defined by an abstract class."[3] The authors note that this principle is a common theme of the design patterns they describe; others have later observed that most classes involved in the patterns are not really classes, but roles.[4]

A suggestion similar to that of Gamma is made by D'Souza and Wills in their Role Decoupling pattern: "Declare every variable and parameter with an abstract type written for that purpose."[5] The rationale behind this recommendation is that only a few classes utilizing other classes (via instance variables) or collaborating with them (in operations) need access to all of the other classes' functionality. Rather, the access can be restricted to the features actually required, and this restriction is best realized by declaring variables and formal parameters as interfaces

rather than classes. Doing so results in the decoupling of classes; it is picked up by UML in the form of interface specifiers (see below) placed at association ends. However, unlike UML and Gamma, D'Souza and Wills explicitly refer to the decoupling interfaces as roles.

The "program to an interface, not an implementation"[3] principle is quite old. By separating the interface specifications of abstract data types from their implementations, CLU (a language from the 70's) allows the abstract data types (called clusters) of a program to be compiled separately and still be statically checked for type correctness. What is rather new, however, is the idea that interfaces might be only partial specifications of classes, specifications that highlight one particular aspect or usage of a class, and that roles are the appropriate conceptual abstraction for this.[6]

Even though it may not be reasonable to introduce a new interface for every variable in use, there are good reasons to divide the total interface of a class into several (possibly overlapping) facets. One is that substitutability of the supplier class (the one whose instances are assigned to the variable) depends on its use. It is the particular usage of the class that rules over which other classes' instances can step in to replace those of the original supplier. Consequently, all and only these classes (together with the original supplier class) should be comprised under one interface, which should be well distinguished from interfaces resulting from uses of the class, and comprising other substitutive classes. For instance, when it comes to making a decision, a person may be replaced by an Artificial Intelligence (AI) program (both playing the DecisionMakingAgent role), and when it comes to signing a contract, a person may be replaced by an organization (both playing the Contractor role). DecisionMakingAgent and Contractor are interfaces or roles of Person, and a variable's declaration as being of one or the other type depends on the particular use of that variable. (In case the variable is used for both purposes, perhaps there should be a common superinterface of the two.) There is a broad span between having one interface per class (to separate interface and implementation) and having one per variable with that class as a supplier; as always, the golden mean lies somewhere in between and depends on the particular problem. As a rule of thumb, designers may be guided by testing whether the interface they are about to introduce conceptually is a proper role of the problem domain.

Other authors also advocate the program-to-an-interface strategy, but have a different understanding of roles. For example, whereas Passenger is typically recognized as a role and Person as its role-player class, Passenger may itself be viewed as a class having instances of its own.[7] A passenger is then represented by a pair of instances, one of Passenger and one of Person. Request to a person as a passenger must be directed to the Passenger instance, while requests to a passenger as a person must be directed to the Person instance. Persons and passengers can delegate tasks to each other, but if one is to substitute for the other, they must both implement the same interface. Although this interface can be the role's, interfaces and roles are not the same concepts.

Still others define a pattern of roles and role models and suggest it mainly as a solution to the aforementioned current OO programming languages lack of support for dynamic and multiple classification.[8] As in the Person/Passenger example above, the pattern separates the role from the class that takes the role, the role model, and represents both as instantiable classes. In addition, role classes can be arranged in a hierarchy (the role hierarchy) and serve as role models for other roles. However, since patterns typically consist of roles,[4] modeling roles as patterns is prone to be circular. Role and role model are themselves roles (of classes); and the fact that a role can be the role model of other roles means this class plays both roles (*role* and *role model*).

## ROLES AND INTERFACES IN UML

In the wake of Java, interfaces have become a prominent OO programming concept. It even appears that interfaces are recognized as the key contribution of Java as a general-purpose programming language. However, to ensure that interfaces are properly designed into OO programs from the beginning, a suitable conceptual abstraction that blends well with other OO modeling concepts is needed. Roles, it seems, are such an abstraction, and although roles and interfaces each have their place in OO modeling, one may argue that the two concepts should really be only one.

The interfaces of UML are largely the interfaces of Java. Roles, on the other hand, are a different concept in UML: They appear as *rolenames* naming the places of relationships (the *association ends* in UML jargon) and as *collaboration roles* representing the participants of a collaboration (interaction). These two notions are inherited from the ER diagram and OO methods such as OORAM,[9] respectively, and are mostly independent of each other.

Figure 1 shows an excerpt of UML's metamodel as compiled from the original specification.[1] For the sake of conciseness, classifiers other than Class and Interface are omitted; the complete list can be found in Rumbaugh.[2] It must be noted, however, that much of UML's complexity (and much of its imprecision) is due to this generalization and the resultant genericity; therefore, the correctness of the following and all other argumentation critically depends on what is under the classifier term. Also notice that AssociationClass, the common subclass of Association and Class has been omitted; although a handy modeling concept, it entails certain consistency prob-

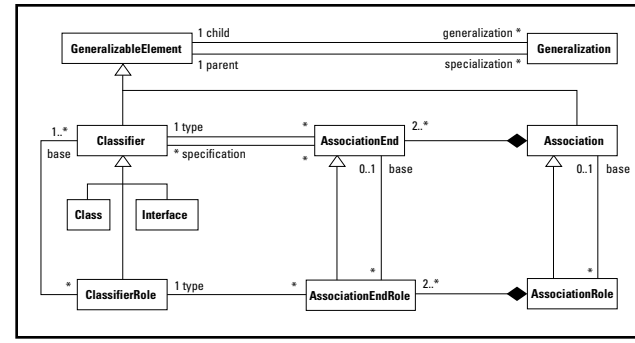


**Figure 1. An excerpt of the abstract syntax forming the structural part of the UML metamodel.**

lems that are not dealt with here.

Quite obviously, numerous constraints are necessary to restrict the possible instantiations of this metamodel to those that make sense. In particular, it must ensure that certain types of classifiers do not appear in certain contexts; for instance, it must be excluded that a classifier role specifies another classifier role as its base, or that an association mixes classes and classifier roles. Of course this is not a real problem, but a number of other things deserve attention.

- The rolename is not an independent concept, but manifests itself in the name attribute of AssociationEnd (not shown), which captures, among other things, multiplicity and navigability information of an association end.
- While an association end itself is not a role in UML, AssociationEnd, as well as Association and Classifier, have specializations that are Roles. Of these, only AssociationEndRole has an (inherited) attribute that is referred to as a rolename.
- Each association end can specify one or more classifiers as *interface specifiers* (mapped to the specification pseudo-attribute; see Figure 1), which restrict access to the instances of the classifier across the association. In a way, these interface specifiers parallel the specification of base classifiers with a classifier role; but despite the symmetric structure of the metamodel, this parallelism is unapparent. Whereas an association end has two ways (namely the type and the specification pseudo-attribute) to constrain the set of instances that can engage in an association, an association end role, which is by definition a restriction of its base association end, has three: two via its base (base.type and base.specification) and one via its associated classifier role (type.base). A fourth, via the interface specifiers that would be inherited from AssociationEnd (specification.base), is dropped, presumably because it is considered colliding with specifying base classifiers. However, inconsistencies between the remaining three paths are possible and must be avoided in order that some instances can actually play the role.
- Interfaces have no specific use in the UML metamodel — instead, the more general Classifier (including classes) is consistently (ab) used to specify interfaces (as interface specifiers and as the bases of collaboration roles) where the use of the Interface concept is in place.

It seems that roles add much complexity to UML, but without roles, UML, like most of its precursors, is not expressive enough to address certain modeling problems. Different occurrences of the same class in one association or collaboration must be distinguished, and roles are the natural concept to do so. Collaborations, but also associations, can have alternative classes connect to one association end; in such cases, an ad hoc specification comprising only the allowed classes is preferable over the specification of a generalization (an abstract class) introduced only for that purpose. Also, in a collaboration only certain aspects of the objects involved are actually required, so specifying a particular class would be overly specific. Indeed, to maintain the genericity of the model, the use of objects in a collaboration should be explicitly restricted to the aspects needed, and instances of different classes should be allowed to substitute for each other as long as they conform to the role specification set by the collaboration. In essence, it appears that program-to-an-interface is also a valid maxim for OO modeling; and classifier roles in collaborations, thus interface specifiers at association ends. The question is: Is the current UML metamodel the best implementation of this maxim?

## THE MERGER

As it turns out, much of the confusion surrounding the role of roles in UML can be avoided by making a few fundamental commitments. These are:

1. The metaclasses Interface and ClassifierRole are merged into a new metaclass Role. The restrictions regarding interfaces in UML (that they cannot have attributes or occur in other places than the target ends of directed associations) are dropped. Class and Role are strictly separated—while classes can be instantiated (unless of course they are abstract), roles can not. Also, classes and roles are generalized separately.
2. The association between classifier roles and their base classifiers is replaced by a new relationship, named populates, that relates classes with the roles their instances can play. (It is convenient to speak of a class as *populating* a role and of an instance as playing a role. It is important that these are distinguished: populating corresponds to the subclass relationship among classes, while playing correseponds to the instance-of relationship of an instance to its class. The diction in the literature is often ambiguous in this regard.)
3. Association ends are required to connect to roles exclusively. Because roles are interfaces and subroles can combine several interfaces, both pseudo-attributes type and specification are replaced by one new relationship, fills, associating each association end with one role. The classes whose instances actually participate in an association are specified only indirectly, via the populates relationship between classes and roles. Association ends need not be given a rolename; if they are, this name must equal the connected role's. Every role must be unique within an association, i. e., no two association ends of one association must specify the same role.
4. AssociationEndRole, AssociationRole, and the generalization of associations are replaced by *association overloading*. For this purpose, a new metaclass, Signature, is introduced whose instances stand between an association and its (overloaded) association ends. Thus, rather than giving rise to the concept of association roles, an association restricted in the context of a collaboration entails a new instance of Signature, comprising new association ends, each connected to a role defined by the collaboration.

The so-changed UML metamodel is shown in Figure 2. First and foremost, it reflects the suggested conceptual equivalence of in-

terfaces and roles. In particular, interface specifiers and classifier roles (specifying interfaces indirectly via their base classifiers) are no longer treated as different concepts, and UML's indifference with regard to a classifier's being a class or an interface is lifted. The association roles and association end roles of UML are not roles in the usual sense (but they share superficial properties with classifier roles, such as being slots in collaborations and having bases); calling them roles is a peculiarity of UML, and not calling them so (due to their abolition) is not likely to be considered a loss.

As a side effect, the suggested changes come with a reduction in the number of modeling concepts, which by itself has a certain value. The only new concept, Signature, should be familiar to both OO programmers and formalists; particularly the latter will have missed it in the UML specification. However, replacing association roles, association end roles, and association generalization by a single concept, namely overloading, may appear inapt. Are association roles and association end roles fully covered by the overloading of associations? And what has association generalization got to do with it? A closer look at the UML specification gives the answer. Given that "association roles specify a particular usage of an association in a specific collaboration," the "constraints expressed by the association ends are not necessarily required to be fulfilled in the specified usage."[1] For instance: "The multiplicity of the association end may be reduced in the collaboration, i.e., the upper and the lower bounds of the association end roles may be lower than those of the corresponding base association end, as it might be that only a subset of the associated instances participate in the collaboration instance."[1] In other words, the extent of an association role is a subset of the extent of its base association.

The UML specification continues: "Similarly, an association may be traversed in some, but perhaps not all, of the allowed directions in the specific collaboration, i.e., the isNavigable property of an association end role may be false even if that property of the base association end is true. [...] The changeability and ordering of an association end may be strengthened in an association-end role, i.e., in a particular usage the end is used in a more restricted way than is defined by the association."[1] But this is precisely what the specialization of an association amounts to: "As with any generalization relationship, the child element must add to the intent (defining rules) of the parent and must subset the extent (set of instances) of the parent. Adding to the intent means adding additional constraints. A child association is more constrained than its parent."[2] Disallowing navigation is more constrained than leaving navigability open (that is, allowing it), and being sorted is more constrained than not being sorted (it is implied by an additional condition, that the elements are ordered).

It seems that association roles are fully covered by the specialization of their base associations. If an association role does not have a base, then it is not a restriction of an association and may be replaced by a plain association (perhaps with its scope limited to the defining collaboration). The difference between association specialization and association overloading (which is not defined in UML) is that association specialization is generally less constrained than overloading and, less importantly, that the specialization of an association may be assigned a different name. Without going into the technical details, these differences are not likely to play a role in practice, since both association roles and association specializations are rarely renamed and the constraints with regard to overloading are usually automatically met, as suggested by Figure 3.

A final note on the representation of generalization in Figure 2. UML introduces Generalization as an instantiable metaclass (see Figure 1). In order to avoid inconsistencies, it must be declared for all generalizable elements what is inherited down each generalization relationship (instance of Generalization). The changed metamodel takes a simpler approach: it represents generalization as an (overloaded) association of the metamodel. Note that generalizes is of the same order as populates, while all instances of Association are of a lower order. This way, no precautions avoiding inconsistencies and paradoxes need to be taken.

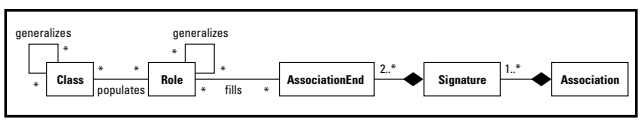


**Figure 2. The new metamodel.**

## WHAT THE MERGER ENTAILS

The merger of role and interface results not only in a reduction of concepts, it also gives interfaces a more prominent status in OO modeling, a status they have long earned in OO design and programming. However, while the changed metamodel may indeed be considered a simplification, it is the modeling language's notation rather than its abstract syntax that must stand the test of practicability. More specifically: not the metamodel, but the diagrams drawn by the modeler must be compact, intelligible, and unambiguous. A closer look at the implications of the changes for the UML diagrams and notation is thus in place.

The class diagram will most visibly be affected by the changes. Interfaces, now termed roles, will be seen more frequently since all associations must end at roles. The rolenames at the association ends, if present, should disappear; their role is taken over by the names of the roles connected to the ends. Self (circular) associations should also disappear, because each role must be unique within an association. Let us look at a couple of examples.

Figure 4(a) shows a simple class diagram. In most cases, it

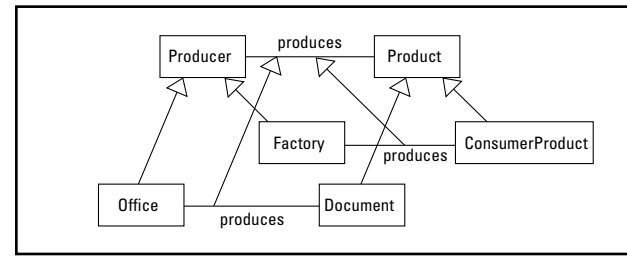


**Figure 3. The specialization of an association in UML. This type of diagram is also characteristic of association overloading.**

would be considered an example of poor design since it models roles as classes; note that the class names would make perfect rolenames for the association ends. Remodeling the class diagram according to the changed metamodel leads to a class diagram like the one shown in Figure 4(b). To make the distinction between roles and classes clearer, the use of circles for roles (as provided for interfaces by the UML specification[1]) is preferable. The populates relationship is depicted as an implements dependency (dashed arrow with closed hollow arrowhead).

Class diagrams like that of Figure 4(a) are seen less and less frequently seen in OO models; the inappropriateness of representing roles through classes is only too apparent (unless the role model pattern is used; see "Discussion," p.31). Instead, the base classes (the ones populating the roles) are viewed as central to a class diagram, and the roles are put in the rolenames of the connected association ends. Figure 5(a) is a typical example of this, where all association ends have (unique) rolenames. However, association ends are not roles, and although rolenames facilitate reading, they add nothing to the structure of a model.

Figure 5(b) contrasts this with a class diagram that has the added structure that conforms to the changed metamodel. The two look very different at first glance, but closer inspection reveals that there is a one-to-one correspondence if only the rolename of an association end is interpreted as the name of an implicit role the class near it implements. Thus, with a little redefinition (and a corresponding adaptation of the mapping rules) the conventional class diagram notation can be kept with the new metamodel.[10]

Although a class diagram with explicit roles looks more crowded, it has certain advantages. First, it allows it that different classes without a common superclass to take the same place of an association, simply by populating the same role. For instance, the role Retailer in Figure 4(b) could be equally populated by class Store; the information can be added by inserting the class in parallel to Person. Second, if interface specifiers restricting the access to a class through an association should be needed, they can be specified by detailing the corresponding role definition. Last but not least, making roles explicit emphasizes the plug points of a model, which are not usually apparent from a role-less class diagram. However, the concomitant loss of connectivity in class diagrams like the one in Figure 5(b) makes intuitive understanding of the model more difficult, and may thus be considered too high a price for the visibility of decoupling and the resultant modularity. In these cases, it is good to know that—under the changed metamodel—a class diagram such as Figure 5(a) is formally equivalent to one with explicit roles, hence supporting decoupling without sacrificing readability.

UML also specifies an object diagram that is rarely used but also affected by the change of the metamodel. Since roles have no instances of their own, but recruit them from the classes populating them, all links (the instances of the associations) between objects would be expected to end at these objects. However, to reflect that an object can only be referred to via the roles it plays, it is appropriate that the links end at role symbols connected to the object. Although UML's lollipop notation is not reserved for the instance level, it is here that its intuitive expressiveness unfolds. The roles an instance plays are indicated by the circles connected to it. The collaboration diagram of Figure 6 gives an impression of what object diagrams with roles may look like.

The next major diagram type affected by the change of the metamodel is the collaboration diagram. In UML, there are two types of collaboration diagrams: one at the instance level and a lesser-known one at the specification level. The former is basically an object diagram enhanced by sequenced method calls. That it is also an instance of a collaboration diagram at specification level is reflected by the classifier role names that may trail the objects' names. According to the change of the metamodel, however,

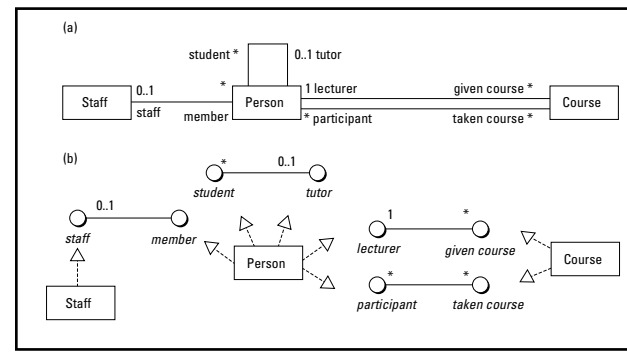


**Figure 5. Alternative class diagrams. (a) is in conventional style using rolenames to label association ends, while (b) has a separate role for every association end. The example is taken from "A Formal Approach to Collaborations in the Unified Modeling Language."**[10]

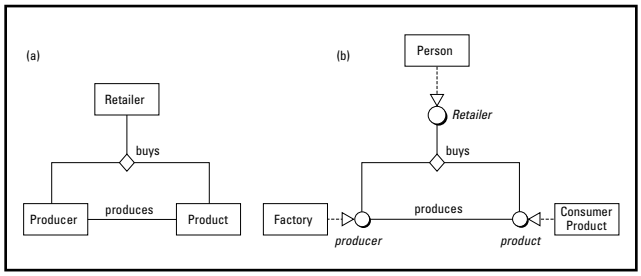


**Figure 4. A class diagram with classes representing roles (a) and its transcription (b), according to the metamodel of Figure 2. Roles are drawn as circles, and the** populates **relation as dashed arrows.**

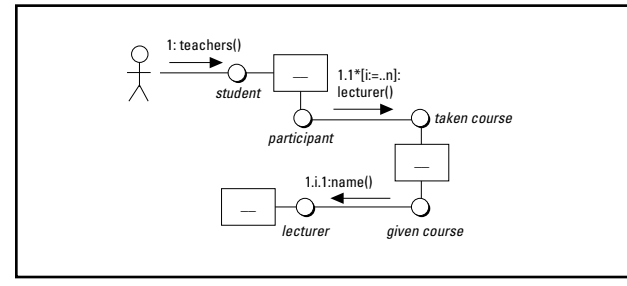


**Figure 6. A collaboration diagram with all objects accessed via roles. Objects remain anonymous and are typed only by their roles. This ensures maximum flexibility.**

classifier roles are ordinary roles the instances play, so no extra notation is needed. A collaboration diagram at instance level drawn accordingly is shown in Figure 6; in that the (anonymous) objects carry no class information conforms to the UML specification of classifier roles (see above) and is in the spirit of plugability.

Collaboration diagrams at the instance level are often considered alternative representations of sequence diagrams. However, since sequence diagrams have only one lifeline per object that, due to the limitations of paper, can only be approached from two sides, roles will have to be added on a call-by-call basis or dropped all together.

The effects of the change in the metamodel on collaboration diagrams at the specification level are bigger. While dropping association roles and association end roles has no visible influence on the representation of collaborations (they are depicted as associations and association ends), the classifier roles are replaced by roles, and their bases (if specified) by the classes populating them. However, the roles replacing the classifier roles are not alone; they are complemented by the roles introduced by the association ends (see Figure 5). Thus, two interpretations of a collaboration diagram like the one in Figure 7 are possible: either the roles attached to the association ends are overridden by the ones corresponding to the classifier roles, or the roles induced by the classifier roles are declared subroles of those at the association ends. The former is a case of association overloading (the restriction of associations to the needs of a collaboration), while the latter is a case of association inheritance (from superrole to subrole). Note that, one way or another, the translated collaboration diagram does not differ syntactically from a class diagram. This is not indicative of a loss in expressive power, but unveils that collaboration diagrams like Figure 7 are not so much interaction diagrams, but an attempt to add more detail to the class diagram of Figure 5(a), detail that can only be specified through the use of roles (see "Discussion"). However, because roles are now an integral part of class diagrams, collaboration diagrams at the specification level (such as in Figure 7) without interaction-specific information (method calls, etc.) do not constitute a diagram type in its own right.[11]

## DISCUSSION

Definitions of the role concept abound in the literature.[11] Many deviate from what has been presented here. In fact, the discovery of roles as a modeling concept usually provokes a stereotypical reaction: Why not model roles as subclasses, multiple roles as their intersection classes, and let the role-playing instances dynamically migrate between the extents of these classes? Indeed, this seems a natural solution, particularly since the roles seem to be more specific than the classes contributing the role players and because the roles' extents (the sets of their instances) appear to be subsets of those of the classes. But even if instance migration were made possible in next-generation programming languages and the hassle of declaring the necessary (combinatorial) number of intersection classes were willingly gone through, one would still be presented with a flawed implementation of the role concept, as the following considerations show.

Every person can, in principle, be a student, an employee, or whatever roles a person can assume at some time in their life. Therefore, roles do not restrict the extents of the populating classes. Rather, because a role like Customer can not only be played by persons, but also by organizations (without one being a subtype of the other), the extents of roles are supersets of those of the populating classes. Besides, if Customer were a subtype of Person and Organization, it would be empty, since the two are disjoint. *Roles are not subtypes, but supertypes;* for a deeper discussion, see "On the Representation of Roles in Object-Oriented and Conceptual Modeling."[11]

Another popular concept of the roles of an object is that of coordinate or adjunct instances, one per role. With this approach it is easy to let roles play roles, and to let each role have its own state. However, it suffers from a serious conceptual inadequacy: an object and that object in a role are different instances and thus have different (object) identifiers. This conflicts with a fundamental assumption of data modeling—that one object of reality should be represented by one object in the model, which in turn was one of the primary reasons for the introduction of roles in the first place.[12] But the objection is not limited to the conceptual level, it also has practical implications. At the very least it requires that all roles have knowledge of the objects they are played by, and that tests for object identity among roles are delegated to the known role players. Although modeling roles as adjunct instances is not completely out of place, with current OO programming languages it has the character of a pattern (like that described by Renouf and Henderson-Sellers[8]), and is not a native programming language construct.

Independently of all pragmatic considerations, there are strong conceptual arguments in favor of the merging of roles and interfaces. For instance, Guarino[13] presents a definition that differentiates roles from natural types by the following two conditions:

1. An instance that belongs to a natural type at any point in time must always belong to that type, under all circumstances and at

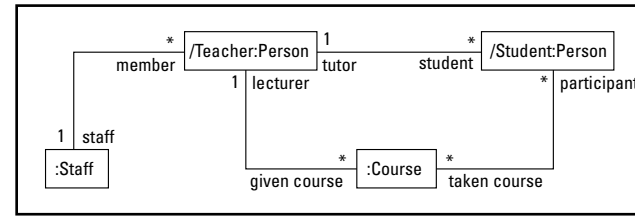


**Figure 7. A collaboration diagram at specification level refining the class diagram of Figure 5 (a). It translates to a class diagram with roles like that of Figure 5 (b) (see text). The example is taken from "A Formal Approach to Collaborations in the Unified Modeling Language,"[10] and "OMG Unified Modeling Language Specification."[1]**

all times. Otherwise, it invariably loses its identity. For roles, this is characteristically not so.

2. A role is characterized by the fact that its instances, the role players, are necessarily related (by another whole/part-relation) to other instances. A son is only a son if some other individual is the father, and a student is only a student if she enrolls in a subject. The instances of a natural type, on the other hand, are instances of that type qua being, not per relatedness to some other instance; a person is a person independently of the existence of any other individual.

Of course, these definitions are not absolute: what is a role in one context may be a natural type in another. However, although ontologically motivated, they fit the software reality of classes as natural types and interfaces as roles. As for the first condition, an instance cannot change its class without losing its identity, but it can take up and drop roles (by being assigned to variables of the corresponding interfaces) as needed. A variable on the other hand, be it an instance variable or a formal parameter, is always an expression of relatedness, so requiring that all variables should be declared as interfaces (or roles) complies well with the second condition from above.

Of course, there are also downsides to the proposed merger. First and foremost, the requirement that all associations must end at roles will inevitably cause an inflation of names. This is particularly disturbing if the names are artificial in character (and the concepts they name close to meaningless), because a model is meant to be a picture of reality, not a complication. There will surely be classes that have only one role, and if this role completely specifies the class, it is difficult to see why there should be two names. One remedy would be to restrict role modeling to the "big" associations and collaborations (the higher levels of abstraction, or analysis) of a system; but if you pick up a recent book on OO design and programming, you will find that interfaces are used very generously, without caring about an inflation of types.[5,7]

Another problem is that defining roles as interfaces does not cover everything one might expect from the role concept. For instance, in certain situations it might be desirable that an object has a separate state for each role it plays, even for different occurrences in the same role. A person has a different salary and office phone number per job, but implementing the Employee interface only entails the existence of one state upon which behavior depends. In these cases, modeling roles as adjunct instances would seem more appropriate but, on the other hand, salary is really an attribute of the job, not of the employee, and the phone number is an attribute of the workplace.

A third problem is that the implementation of classes populating many roles, such as Person, will become very large. Although several workarounds are thinkable,[14] one should not forget that natural persons *are* complex entities and that roles structure the interface of such entities in a natural way. Besides, the change of state of an instance in one role often affects other roles, a dependency that can be implemented as a side effect if everything happens in one place.

A number of issues remain. For example, it must be checked how much of the UML metamodel's genericity can be retained under the given redefinition. Does the metamodel of Figure 2 also work for use cases and other classifiers? Even if this is not the case, one may ask whether concepts as diverse as use cases and classes can be comprised under one abstraction that can be used consistently across all purposes. I doubt it.

## CONCLUSION

Equating interfaces with roles gives a proven OO programming and design construct a meaningful conceptual representation. With a few additional commitments, the number of elementary modeling concepts can be considerably reduced, resulting in a simpler metamodel structure on one side and in a clearer separation between structure and interaction diagrams on the other. ■

## REFERENCES


1. *OMG Unified Modeling Language Specification Version 1.3.* June 1999. See www.omg.org.
2. Rumbaugh, J., I. Jacobsen, and G. Booch. *The Unified Modeling Language Reference Manual,* Addison-Wesley, Reading, MA, 1998.
3. Gamma, E., et al. *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, Reading, MA, 1995.
4. Buschmann, F. "Falsche Annahmen (Teil 2)," *OBJEKTspektrum,* 84-85, April 1998.
5. D'Souza, D.F. and A.C. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach,* Addison-Wesley, Reading, MA, 1999.
6. Firesmith, D.G. and B. Henderson-Sellers. "Upgrading OML to version 1.1: Part 2—Additional Concepts and Notation," *JOOP* 11(15): 61–67, Sept. 1998.
7. Coad, P., et al. *Java Design: Building Better Apps and Applets 2nd ed.,* Prentice Hall, Upper Saddle River, NJ, 1999.
8. Renouf, D.W. and B. Henderson-Sellers. "Incorporating Roles into MOSES" *Proceedings of the 15th International Conference on Technology of Object-Oriented Languages and Systems: Tools 15,* C. Mingins and B. Meyer, Eds., 71-82, Prentice Hall, Englewood Cliffs, NJ, 1995.
9. Reenskaug, T., P. Wold, and O. A. Lehene. *Working with Objects—The OOram Software Engineering Method,* Addison-Wesley/Manning, 1996.
10. Steimann, F. "A Radical Revision of UML's Role Concept," *UML 2000. Proceedings of the 3rd International Conference,* A. Evans, S. Kent, and B. Selic, Eds., 194-209, Springer, 2000.
11. Övergaard, G. "A Formal Approach to Collaborations in the Unified Modeling Language," *UML '99 LNCS 1723,* Springer, 1999.
12. Steimann, F. "On the Representation of Roles in Object-Oriented and Conceptual Modeling," *Data & Knowledge Engineering,* 35(1):83–106, 2000.
13. Bachman, C. W. and M. Daya. "The Role Concept in Data Models," *Proceedings of the 3rd International Conference on Very Large Databases,* 464–447,1977.
14. Guarino, N. "Concepts, Attributes and Arbitrary Relations: Some Linguistic and Ontological Criteria for Structuring Knowledge Bases," *Data & Knowledge Engineering,* 249–261, Aug.1992.
15. Fowler, M. "Dealing with Roles," *Supplement to Analysis Patterns: Reusable Object Models,* Addison-Wesley, Reading, MA, 1996.


**Friedrich Steimann** is a Lecturer in Applied Informatics at the Universität Hannover in Hannover, Germany. He has specialized in Computational Linguistics, Medical Informatics, Software Engineering, and has been involved in commercial software projects of all sizes. His current work focuses on OO software modeling. He may be reached at steimann@acm.org.