# Role + Counter Role = Relationship + Collaboration

## Towards a Harmonization of Concepts

Friedrich Steimann

Lehrgebiet Programmiersysteme
Fakultät für Mathematik und Informatik
Fernuniversität in Hagen
D-58084 Hagen
steimann@acm.org

## Abstract

Common conceptions of the notions of role, relationship, and collaboration in object-oriented programming and related disciplines are briefly recalled and related to each other, working towards a view that allows it to fit them all together, eventually setting the cornerstones for the definition of a truly useful and widely accepted novel programming language construct.

## 1. Introduction

More than 30 years after its inception, the entity-relationship (ER) diagram [9] is still a prevalent modelling notation. In fact, even though object-oriented programming builds on the navigational [2] rather than the relational data model [10], the most popular object-oriented design notation, the UML class diagram, with its associations very much resembles the ER diagram. However, there is a conceptual gap between the associations of class diagrams (henceforth referred to as relationships for the sake of uniformity) and the pointers of object-oriented programs implementing them. So-called model-driven engineering, which needs to close this gap, provides transformations mapping relationships to stereotypical code fragments often taken from patterns [13], implementing the semantics of the relationships. However, from a programming language designer's viewpoint, such code is a nuisance.

As has been suggested before, the dyad of entity and relationship is naturally complemented by a third concept, that of a role [16]. Not only in the realm of software, roles are considered placeholders in relationships or collaborations [15] that specify the required properties of role players, objects participating in a relationship in that role. As such, they are abstract specifications or types, abstract in the sense that unlike classes, they do not specify concrete entities (and therefore cannot be instantiated). In fact, different classes can provide players for the same role, provided that they all comply with the requirements of the role type. This makes roles akin to interface types of JAVA-like languages [17].

Like relationships, roles have found their way into object-oriented programs mostly in the form of patterns. In fact, despite the strong arguments for the role-as-interface view (which would suggest that JAVA et al. already offer roles as a language construct), the role object pattern [4] seems to be the most popular implementation of roles in object-oriented languages today.

The following discourse suggests reconsidering common notions of role, relationship, and collaboration so as to harmonize their definitions in such a way that a single new programming language construct can be devised that serves all three. I do not attempt to present such a language construct here, but rather wish to share my insights and believes about possible cornerstones of its definition. For this reason (and also because of the spatial restrictions imposed for this clearly unfinished work) I refrain from showing syntax, let alone precise semantics, of my proposals. My contribution may therefore be seen as an addition to [12].

## 2. Roles

The concept of roles has been discussed broadly in the literature. The properties regarded as characteristic for the role concept vary from author to author, with some of them contradicting [16]; nevertheless, a few core properties seem to be largely unquestioned. This includes the property that a single object can play several roles of different or the same kind both simultaneously and sequentially, and that the same role can be played by different objects of the same and different kinds. Raised to the type level, this means that the relationship between role types and class types (as sources of role players) is generally *m*:*n*.

### 2.1 Roles and behaviour

A role elicits the behaviour of an object in a given context. For instance, the role of father specifies behaviour expected from a man in the context of his fatherhood. On an abstract level, the role-specific behaviour is specified independently from the (nature of the) role player; in general, the class of the role player should be substitutable by another without compromising the role specification.

The view of roles as abstract behaviour specifications suggests that roles can be implemented as JAVA style interfaces [17]. A class implementing such an interface declares to conform to the role specification the interface represents, i.e., that its instances can play the role. A variable (or attribute in modelling terms) declared with an interface as its type points to a role player instance the precise class of which is unknown; this is no problem, since the interface captures everything that is needed to be known from the object in the context in which the variable is declared. Note how this complies with the idea of variables implementing relationships [13], with the special addition that the place of the relationship represented by the variable is typed by a role rather than an entity type.

Viewing roles as interfaces and thus supertypes of classes allows it that objects of different classes (unrelated by inheritance) can play the same role. For instance, both class `Person` and class `Computer` can (deliver instances that) play the role of `Teacher` if they both implement the corresponding interface. This allows the computer to fulfil his teaching obligations in quite different ways than a person would do, which seems to model reality adequately. On the other hand, there may be congruence in the behaviour of

computers and persons as teachers, so that it would be nice were one allowed to specify some behaviour, even if only skeletal, as part of the role, with concrete role players filling in the missing (and player-specific) parts. This could be achieved, for example, by allowing template methods in the role specifications, which would then have to be represented by abstract classes.

### 2.2 Roles and state

The role-as-interface view is often challenged by the observation that roles appear to have role-specific state. And indeed, it seems reasonable that a person in the Employee role has some role-specific data such as (office) telephone number and salary. Even if this state were made part of the role-as-type definition (replacing interfaces by abstract classes as role representations), this would not allow a single person to have several office phone numbers and salaries at the same time (namely one per employment). This is a problem since as noted above, being able to play the same role multiply at the same time is a defining characteristic of the role concept (cf. above). The role object pattern solves this problem.

### 2.3 The role object pattern

The role object pattern [4] represents the roles an object plays by adjunct instances, the role objects. Each role object is an instance of a certain class, its role type. The role type specifies role-specific state and behaviour; by instantiating the same role type several times and adjoining the instances to the same role player object, the problem of multiple phone numbers and salaries is solved. At the same time, the role class can implement role specific behaviour independently from the role player class, and forward player-specific requests to the player (cf. the mentioning of the template method above).

Despite its popularity, the role object pattern has some problems. First, it is implemented as pattern, i.e., it requires stereotypical code (such as that for adding and removing roles to and from a role player) programmers would like to avoid. Second, a role player and its roles are different objects with different identities, leading to the problem of object schizophrenia (a conceptual entity with split identities). Last but not least, the role object pattern itself comes with at least two roles, the RolePlayer role and the Role role, which cannot be implemented using the role object pattern, leaving a yawning gap in the semantics of the concept.

### 2.4 Role and counter role

Intuitively, a role is meaningless without one (or more) counter roles, the one(s) with which its player interacts and for which the associated behaviour is required. Role and counter role(s) together form an interaction pattern — sometimes called collaboration [15] —, with parts of the behaviour being specific to the interaction, and parts being contributed by the role players.

The notion of counter role implies that role players occur in pairs (or, more generally, tuples). As will be shown next, pairs (tuples) of roles form a relationship, which can also be associated with state and behaviour. This doubling offers an opportunity for replacing the role object pattern with a native programming language construct that does not suffer its problems.

## 3. Relationships

The concept of relationship is closely related to the mathematical notion of relation.[1] Like a relation, a relationship has an arity *n* and as many different places, each associated with a type that constrains the set of elements allowed to take each place. In mathematics, one distinguishes between the relation signature (or declaration) giving its name and the types of its places, and its extension, a set of tuples formed of members of the types; in data modelling, one distinguishes between a relationship type (sometimes also referred to as its intension) and relationship instances (the whole set of which is referred to as the relationship's extension). In physical systems, the (dynamic) extension of a relationship breathes, i.e., it grows and shrinks as tuples are added and deleted [18]. Both relationships and (mathematical) relations can be restricted by constraints; for relationships, so-called cardinalities (or multiplicities in UML jargon) are in frequent use, for relations, properties such as symmetry or transitivity are common.

### 3.1 Relationships and roles

Ever since Codd's definition of the relational data model [10], places of relationships have been associated with role names to distinguish places (columns) of the same type. Variations of this data model allowed union types (disjunctions of two or more types) to fill the places of relationships, so that a single place could be occupied by objects of various types (a necessity that was already recognized by the network data model [1]). However, these type unions remained ad hoc — in particular, no new type was defined and given the name of the role, as suggested by the role-as-interface interpretation discussed above (where an interface is interpreted as the union type of its implementing classes). Roles as types were first suggested by Bachman in his role data model [3, 19], which was intended to supersede the network model[2].

### 3.2 Relationships and state

In the ER data model [9], relationships do not only link entities, they may have state (attributes) of their own. For instance, an Employment relationship may come with attributes for an (office) telephone number and a salary. And indeed, it seems that conceptually, both attributes are attributes of the relationship, namely Employment, rather than the employee; after all, they could equally well be ascribed to the employer, who pays the salary and owns the phone.

So rather than associating state with one or the other role player, it can be captured as part of the relationship. With the availability of correspondingly conceived relationship types, role types can become stateless, so that the protocol specifications (interfaces) suggested in Section 2.1 suffice.

### 3.3 Relationships and behaviour

Pure data modelling is usually not concerned with behaviour; neither entity types nor relationships types are therefore associated with such. The behaviour of databases is usually restricted to the growing and shrinking of extensions; it is specified using a data manipulation language (DML) which operates over the data structures, but does not associate behaviour with the entities contained therein.

While object-oriented programming enhanced the notion of entity type by adding behavioural specifications (termed methods) to form classes, for some unapparent reason it did not attempt the same for relationship types; rather, it has abandoned them altogether. With one notable exception.

---

[1] Codd defines relationships as the domain-unordered counterparts of (mathematical) relations [10].

[2] and which was devised in parallel to the ER data model (personal communication with Charles Bachman)

### 3.4 Transient relationships: procedures

An often overlooked materialization of relationships in programs are procedures: a procedure head with *n* parameters specifies an *n*-ary relationship whose places are constrained by the procedure's parameter types. The name of the procedure is the name of the relationship. In [18], these incarnations of relationships are termed *procedural*, as opposed to the *structural* relationships known from data modelling.

The greatest limitation of the view of procedures as relationships is that their instances are necessarily transient. More concretely, relationship instances are created as activation records on the call stack and therefore exist only as long as the corresponding procedure is executed. A corollary to this fact is that these relationships cannot be queried programmatically (unless the program has reflective access to the call stack). Instead, the (temporary) relationship between the actual parameter objects is exploited by the procedure body, which associates behaviour with the relationship specified in the procedure head, behaviour which is automatically executed whenever a relationship instance is created (the procedure is called). Therefore, relationships not only already exist in object-oriented programming languages, they also have what relationships from data models are missing: associated behaviour.

The interpretation of procedures as relationships is perhaps more obvious in logical programming languages such as PROLOG, in which a predicate can be interpreted as a relationship (in the database sense) and as a procedure (in the procedural programming sense). Predicates with ground terms as arguments (i.e., not containing variables) are considered facts and represent persistent relationships instances; those with non-ground terms (containing variables) represent transient relationship instances which are computed during the solution of a goal (by assigning the variables values temporarily). Transient relationship instances can be turned into persistent ones using PROLOG's metaprogramming facilities; thus, in PROLOG relationships represent data *and* specify behaviour. This seems like a desirable property for relationships embedded in an object-oriented programming language.

## 4. Harmonization of concepts

It seems that the terms relationship, role, and collaboration are well-established in the thinking of software engineers, yet their exact definitions and the preferences of one over the other vary from individual to individual. However, given a few fundamental (and presumably also widely accepted) assumptions, namely that

- a role can specify role-specific state and behaviour,
- a relationship can specify state (the relationship types of the ER data model) and behaviour (procedures) associated with the relationship,
- roles come in pairs (or, more generally, tuples) of counter roles,
- each place of a relationship is associated with one role, and that
- collaborations specify interactions among roles,

I suggest to harmonize the definitions of role, relationship, and collaboration so that

a) a relationship type specifies the role types whose implementing classes contribute to it,

b) the relationship captures the states associated with the contributing roles,

c) each role specifies the abstract behaviour required from its role players independently of any counter roles, and that

d) collaborations associated with a relationship capture that part of the behaviour that deals with the interactions among role players in the context of the relationship, but not their individual contributions to this interaction.

A relationship with collaborations is thus defined by

1. an set of role types specifying the abstract behaviour required from the role players, but not their role-specific state,
2. a set of attributes specifying the state associated with the relationship or any of its roles, and
3. a set of behavioural specifications (called collaborations) that have access to the relationship state and whose implementations can resort to the behaviour associated with the relationship's roles.

This definition of relationships caters for two use cases:

1. Structural relationships: This covers persistent relationship instances that, like objects, exist from instantiation to disposal (by a delete operator or by un-assignment and subsequent garbage collection). It is a question of language design whether such relationship instances have identity and can be assigned to variables, or are simply elements of a (program-accessible) relationship extension in which each tuple is uniquely identified by the combination of its role-playing objects.[3] In either case, relationship extension manipulation occurs through language constructs analogous to those of a standard DML, which includes operations for adding, deleting, and updating tuples.[4] Also, relationship extensions can be queried for selecting tuples that satisfy certain conditions; collaborations will always be executed on the result sets of such queries (possibly singletons), using the role players and relationship state contained in each tuple as implicit parameters to the collaboration.
2. Procedural relationships: This allows ad hoc collaborations, i.e., collaborations of objects without creating a corresponding persistent relationship instance. As suggested in Section 3.4, when a method is called relationship instances are created on the fly; for this, however, the elements of the relationship instance (tuple) must be supplied with the invocation (as the actual parameters of the collaboration; note that these are the same parameters that are implicit for structural, or persistent, relationship instances).

Note that in either case, collaborations are called on a relationship type; in particular, their invocations are not dynamically dispatched on relationship instances (as long as no relationship subtyping is defined, such a dispatch does not make sense). This offers an interesting opportunity: the dispatch of collaborations on its participants' role types. Since all participants are treated symmetrically, this would amount to a multi-dispatch; however, such a dispatch requires role subtypes and covariant redefinition of relations, which are not considered here (but see [8, 18, 20] for brief discussions of the possibilities). It is interesting to note that this kind of multi-dispatch does not break the encapsulation of the parameter (i.e., role-playing) objects as long as access of the

[3] The latter would avoid dangling pointer problems that occur once relationship instances are explicitly deleted (cf. the discussion in [8]).

[4] Note that it is another language design decision whether updates are restricted to attributes or whether role players can be exchanged (where the former means that role-players are considered the primary keys of the relational calculus; cf. the discussion in [7]).

collaboration implementation to role-specific (now: relationship-specific) state suffices.

Another note is that I left out the description of a DML intentionally; the possibilities are well-explored in the literature. It should be kept in mind, though, that providing highly usable queries, or iterators, is a key to the expressiveness of a language (one of the lessons taught by the success of SMALLTALK).

## 5. Discussion and related work

When introducing relationships as a first class programming construct to object-oriented programming languages, one is immediately confronted with the question, in which respect should this introduction be different from, or go beyond, the added expressiveness offered by embedded SQL or the integration of any other standard relational calculus? A corollary to this question is whether re-implementing a relational calculus in an object-oriented programming language adds some value beyond the syntactical integration of two formalisms. My feeling is that an "orthogonal" addition misses the chance of rethinking the object and the relational paradigm, and that a well-devised extension can indeed be something rather different than the plain sum of existing standards.

OBJECTTEAMS/JAVA [11] extends JAVA with an implementation of the role object pattern using aspect technology. It employs a type-level relationship playedBy which relates role to class types. So-called teams represent contexts, or relationships, which are defined as an interaction of roles and in which the participating objects are always represented by corresponding role instances. In JAVA terms, teams are special kinds of classes that define their participating roles as inner classes. Together with POWERJAVA [5] OBJECTTEAMS/JAVA is perhaps the most advanced implementation of relationships in object-oriented programming to date.

In their JAVA-based calculus RELJ, Bierman and Wren define (binary) relationships as class-like types with fields and methods as members [8]. Each relationship type specifies two classes as its participants (the types of the relationship's places), and optionally one relationship type as its supertype. Relationships can be instantiated; like with classes, the instances are represented by pointers that can be stored in variables. This is at conflict with explicit relationship instance deletion, a problem that the authors have postponed until more experience has been gathered. From their description, it is unclear how relationship methods are handled, in particular whether and how the role players can be accessed from the body of a method defined in a relationship type.

The relationship aspects of Pearce and Noble use the intertype declarations of ASPECTJ to inject the bookkeeping necessary for maintaining relationships between objects [13]. The relationships themselves are coded as aspects that can carry relationship-specific behaviour. Class definitions may remain ignorant of the relationships for which they supply the participants, which is considered an increase in the separation of concerns. This increase is however immediately lost once one object needs access to another to which it is linked only via a relationship.

Balzer et al. present a formal model for relationships as first-class programming constructs that can interpose role-specific fields (and methods?) into classes [7]. These interpositions resemble the introductions (inter-type declarations) of ASPECTJ (which have been used in the relationship aspects described previously, albeit for a different purpose) in that the introduced fields are instantiated once per role player object, not once per role (participation in the relationship). If the latter is required, the attribute can be defined as a member of the relationship (although this case is not considered).

Like the present work, Baldoni et al. have combined the concepts of roles and relationships in object-oriented programming [6]. However, they have done this by extending the relationship-as-attribute pattern [13] with the role concept of POWERJAVA. The present work rather combines the relationship object pattern with roles, mostly by replacing the participating classes with role types.

## 6. Conclusion

Although faintly present in the form of methods, most contemporary object-oriented programming languages come without explicit language support for relationships, collaborations, or roles. Patterns of implementing these concepts therefore prosper. However, frequent resorting to patterns is usually indicative of a certain language construct missing. Relationships are a hot candidate.

## References


[1] CW Bachman "Data structure diagrams" *SIGMIS Database* 1:2 (1969) 4–10.

[2] CW Bachman "The programmer as navigator" *Commun. ACM* 16:11 (1973) 653–658.

[3] CW Bachman, M Daya "The role concept in data models" in: *Proc. of VLDB* (1977) 464–476.

[4] D Bäumer, D Riehle, W Siberski, M Wulf "The role object pattern" in: *Proc. of PLoP* (1997).

[5] M Baldoni, G Boella, LWN van der Torre "powerJava: ontologically founded roles in object oriented programming languages" in: *Proc. of SAC* (2006) 1414–1418.

[6] M Baldoni, G Boella, L van der Torre "Relationships Meet Their Roles in Object Oriented Programming" in *FSEN* (2007) 440–448.

[7] S Balzer, TR Gross, P Eugster "A relational model of object collaborations and its use in reasoning about relationships" in: *Proc. of ECOOP* (2007) 323–346.

[8] GM Bierman, A Wren "First-class relationships in an object-oriented language" in: *Proc. of ECOOP* (2005) 262–286.

[9] PP Chen "The entity-relationship model — toward a unified view of data" *ACM Trans. Database Syst.* 1:1 (1976) 9–36.

[10] EF Codd "A relational model of data for large shared data banks" *Commun. ACM* 13:6 (1970) 377–387.

[11] S Herrmann "A precise model for contextual roles: The programming language ObjectTeams/Java" *Applied Ontology* 2:2 (2007) 181–207.

[12] S Nelson, DJ Pearce, J Noble "First-class relationships in object oriented programs" in: *The Popularity Cycle of Graphical Tools, UML, and Libraries of Associations* Workshop @ OOPSLA (2007).

[13] J Noble "Basic relationship patterns" in: *Proc. of EuroPLOP* (Addison-Wesley 1995).

[14] DJ Pearce, J Noble "Relationship aspects" in: *Proc. of AOSD* (2006) 75–86.

[15] T Reenskaug *Working With Object: The OOram Software Engineering Method* (Prentice Hall 1995).

[16] F Steimann "On the representation of roles in object-oriented and conceptual modelling" *Data Knowl. Eng.* 35:1 (2000) 83–106.

[17] F Steimann "Role = Interface: a merger of concepts" *Journal of Object-Oriented Programming* 14:4 (2001) 23–32.

[18] F Steimann, T Kühne "A radical reduction of UML's core semantics" in: *Proc. of UML* (2002) 34–48.

[19] F Steimann "The role data model revisited" *Applied Ontology* 2:2 (2007) 89–103.

[20] F Steimann, T Kühne "Piecewise modelling with state subtypes" in: *Proc. of MoDELS* (2007) 181–195.