Applied Ontology *to appear* (2007) 1–25
IOS Press

A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java

Stephan Herrmann
Technische Universität Berlin, Germany

Abstract. A number of proposals exist how to support the concept of roles at the level of programming languages. While some of these proposals indeed exhibit very promising properties, the concept of roles has not found its way into mainstream programming languages. We argue that this is due to the richness of the concept of roles and the fact that each existing proposal focuses on some aspects of roles while neglecting others. We present the programming language ObjectTeams/Java and using the categories of Steimann (2000) we demonstrate that this language covers more aspects of roles than previous approaches. We suggest that a thoroughly defined programming language featuring roles may contribute to a better understanding also in other fields using roles.

Keywords: Roles, context, conceptual modeling, programming language, behavior, method dispatch.

1. Introduction

Conceptual modeling has a significant history of using "roles" as a fundamental means for capturing the relevant entities within a domain Kristensen & Østerbye (1996). The concept of roles also appears in a variety of other research areas such as design patterns and programming language design. While the concept is widely accepted for conceptual modeling, other areas are more reluctant. Such reluctance is in part motivated by the lack of a precise understanding of what roles actually are.

This paper gives a definition of roles from a programming language perspective. In order to structure the discussion we will first present a system of coordinates which allows to group the many properties that are ascribed to roles (Sect. 2). We will then present a programming language with explicit support for roles, ObjectTeams/Java, and relate this language to previous work on roles. We argue that the design of a programming language with explicit support for roles can be seen as a precise foundation of the concepts involved. While formal specifications could equally be used for a precise foundation a real programming language has the advantage of practical applicability. Our definition of roles can directly be validated by writing programs in ObjectTeams/Java and evaluating the results along the following questions: (1) Is it possible to model given domains using ObjectTeams/Java (OT/J for short)? (2) Does OT/J support "natural" models, i.e., can a correspondence between the mental model of a domain expert and a program in OT/J be established and maintained throughout development? (3) Does the resulting program behave in ways that are conform to the given domain?

If the above questions can be answered affirmatively we conclude that the language OT/J could be seen as an ontology, defining in our specific case fundamental concepts "role", "team" and their various static and dynamic relations.

With our presentation we try to convince the reader that it is possible to define the concept of roles in a way that is not only rigorous but indeed precise enough to give actual implementations that directly exploit the concept of roles in all phases of the software life-cycle. We hope that a clear understanding of roles at the implementation level will also help to eliminate the confusion that arises from different interpretations of the concept of roles.

The paper is structured as follows: Section 2 introduces a system of coordinates for describing role-based approaches. We will then present ObjectTeams/Java along those coordinates (Sect. 3).

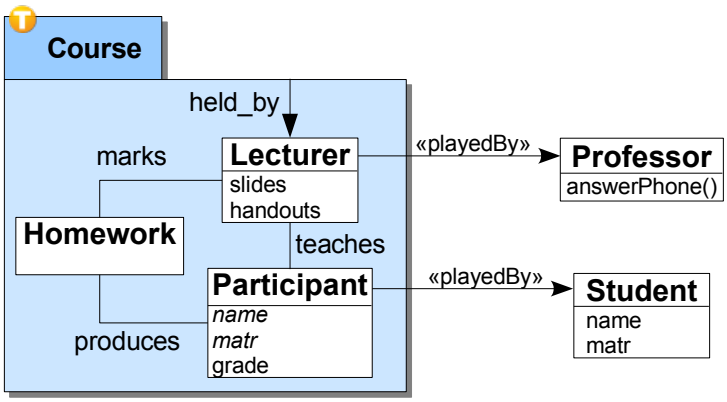

Fig. 1. Example: Roles represent entities within a context

When all relevant features of the language have been presented we will show by means of a simple example (Sect. 4) how the concepts of Object Teams relate to the concept of relationships. Section 5 presents a meta model of Object Teams, thereby summarizing core structural relationships and also discusses possible combinations of concepts. We then apply the criteria defined by Steimann (2000) to ObjectTeams/Java, concluding a nearly complete coverage of all desirable properties (Sect. 6). We conclude with a discussion of related work (Sect. 7).

2. System of coordinates for roles

At a very general level it is fairly easy to establish agreement about the concept of roles. Abstractly spoken, *role* is a relative concept that defines the intersection of an entity and a context of interaction. *Entities* may exist outside the context and may have a well-defined set of properties. Describing on the other hand a *context* focuses on the collaboration of some "elements". These elements can be identified as *roles* which represent existing entities within a given context. The same concept of context is sometimes called a "collaboration", "subject" or a "use case" etc.

Figure 1 depicts a standard example, which will be used throughout this paper. It shows two basic entities, Professor and Student which have no direct association. Also shown is the context Course, comprising a Lecturer and a number of Participants being taught by the lecturer. The context is decorated with a "T" icon, which will be interpreted as a *"team"* when moving to the concrete model of Object Teams in Section 3. In this example Lecturer and Participant are roles of their respective base classes Professor and Student. This relationship is shown by the association with stereotype «playedBy». These roles only exist within the context of a Course. Further details will be explained as we go.

In order to precisely define the notion of roles, it is necessary to specify the semantics of both relationships: the playedBy relationship between a role and its base and the containment relationship between a context and its roles. The literature shows that such precision is not trivial to achieve.

Steimann (2000) has presented a list of 15 features by which roles have been characterized in different approaches. This list of features can be seen as the union of criteria presented in previous approaches. This paper will investigate the concept of roles along a two-by-four matrix, which can basically be seen as an abstraction over Steimann's 15 features.

On the one axis of our system of coordinates we distinguish a **static** and a **dynamic** view. The static view will discuss the structural relationships between concepts. By contrast, the dynamic view describes the (run-time) evolution of structures and possible behaviors and interactions. Along the other axis we will discuss the elements of relevance which are **instances**, **fields**, **methods** and **context**.

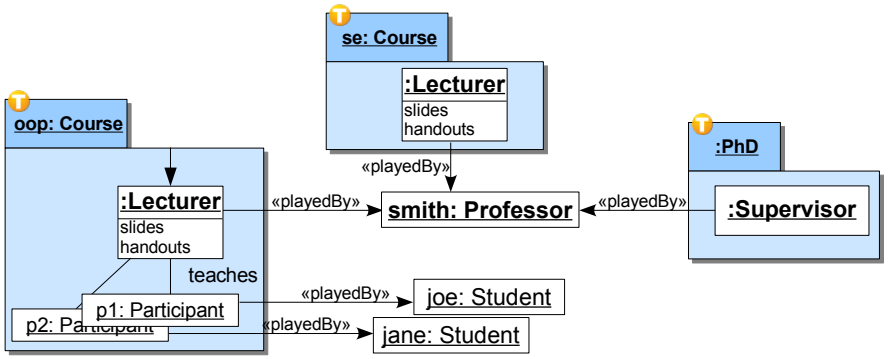

Fig. 2. Instances of the model in Fig. 1

We will now present the relevant properties and design options that can be made explicit using this system of coordinates. This section does not include any judgment which of these design option we actually favor. The following section will then populate the design space by presenting the programming language ObjectTeams/Java, which supports most of the design options presented, whereas a few options where intentionally dropped. We will mark and discuss those restrictions as we go.

2.1. Instances

In contrast to purely class-based approaches, role-based approaches put more emphasis on *instances*. However, no question regarding roles seems to be more controversial than the issue of roles instances and their identity: are roles distinguishable instances with unique identity or are a role and its base the same object? For a discussion from a database perspective, e.g., see Wieringa & de Jonge (1995). Clearly a model supporting roles as distinguishable instances provides greater flexibility, because such a model admits dynamic structures that can not be captured otherwise. On the other hand some authors claim that admitting role instances results in undue complications which should simply be avoided by a more restrictive model of roles. Examples for this restrictive approach are VanHilst & Notkin (1996) and Steimann (2001).

The difference can be illustrated by an instance graph that could be created from the class diagram shown in Figure 1. The center of Figure 2 shows an instance of class Professor called "smith". He plays the role of the Lecturer for each of the Courses "oop" (Object-Oriented Programming) and "se" (Software Engineering). Within an unnamed PhD context the same Professor plays the role of the Supervisor. While all role-based approaches admit the coexistence of a Lecturer role and a Supervisor role for the same base entity (Professor), the existence of two Lecturer roles of the same Professor is problematic (to say the least) in restrictive approaches, i.e., approaches where roles have no identity of their own. We will discuss the options and consequences below.

Our own approach combines the best of both worlds, which can only be achieved by first admitting role instances for the sake of flexibility and then addressing the supposed extra complexity. We will come back to this issue at the end of our presentation of ObjectTeams/Java in Section 3.6, where we try to show that it is actually a non-issue by discussing a solution which allows both interpretations within the same model.

Static view

Assuming roles are instances which can be referenced, a number of structural patterns becomes possible which seem to be characteristic for roles. First of all, it is always the roles which are attached to a base object. The latter remains independent of any roles. This is illustrated by the direction of the «playedBy» relationship. For the separation of concerns it is even crucial to be able to reason about base objects while completely ignoring any roles.

It is a common understanding that a role is always the role of exactly one base object. Looking the opposite direction two kinds of multiplicities come into focus: several roles can be attached to the same base object, namely several roles of different role types as well as several instances of the same role type (cf. Fig. 2).

An additional situation is shown by the *unbound* role Homework. This role clearly is part of the Course collaboration. However, there exists no corresponding concept outside the Course context, so this role is not played by any base class. As such, unbound roles appear as normal classes, except for being participants in a context-defined collaboration.

In the literature *foundation* or *dependence* are considered an essential property of roles (e.g., Masolo et al. (2004); Loebe (2007)). In this sense unbound roles only depend on an enclosing context, whereas bound roles additionally depend on a player.

Dynamic view

With roles being instances attached to a base object it is possible that the life-cycle of a role differs from the life-cycle of its base. At any time during the base object's life-cycle it is possible to attach a role instance to this base. The same holds for removing a role from its base. In some approaches it is even possible to handle (temporarily) unattached roles or change the attachment by moving a role from one base object to another (sometimes called role *migration*).

Using the Course example the issue of unattached roles would translate to a Lecturer who operates on his own, without any Professor playing this role. Since this is not possible it is not clear how unattached roles should be supported in a sound model. Significant additional efforts would be needed, like ensuring that a Course is not "active" as long as its Lecturer role is unattached. We have not found any explicit support for effectively suspending collaborations in the literature.

By contrast, the following questions outline contingent options: Can a Professor become the Lecturer for a new (an existing) Course? Can a Professor put down a Lecturer role? Can the same Lecturer (while the course is running) be played by different Professors in sequence?

The above options for a role's life-cycle are naturally realized using a unidirectional link from a role to its base object, meaning that a role knows its base but the base does not explicitly keep track of all its roles. This raises the question what kinds of navigation are supported between roles and bases.

It is trivial to allow for navigation from a role to its base (using the mentioned base link). This navigation answers questions like "Which Professor is playing the Lecturer role for Course 'oop'?". Conversely, it may also be interesting to ask, e.g., "Which Courses does 'smith' teach?". The latter question asks about the roles of a given base object. This requires either the ability to navigate the role-base link in a backward direction or knowledge about all Courses in the world, and checking if any of their Lecturer roles are played by "smith". Less problematic are requests like: " 'smith', could you please give me your handouts for 'oop'?" (which requires to know "smith's" Lecturer role in "oop"). In that case the Course is known and the corresponding Lecturer role can be determined as the said intersection of "smith" and "oop".

2.2. Fields

In some applications of the concept of roles, significant focus lies on information modeling. For that reason it is crucial to discuss how roles may contribute to the overall system state.

Static view

The observable state of a role is usually defined by the union of its direct fields and the fields of its base object: both sets of fields contribute to the observable behavior of the role. On the one hand it is a central property of roles to *share* the state of their base object. On the other hand an object may have a field only in the context where it plays a specific role. Given the multiplicities discussed above an object may even have different values for the same attribute in different contexts, where these values are actually stored as fields of the respective roles. As an

example consider the fields `slides` and `handouts` in Figures 1 and 2. The same Professor "smith" will have different decks of slides for his Courses "oop" and "se".

This replication of fields being (indirectly) attached to the same base object is not possible without some kind of role instantiation. Without role instantiation a base object would have to grow indefinitely when an unbounded number of roles is attached to it. Furthermore consider the additional field `handouts`: If a Professor (the base object) would manage sets of slides and sets of handouts, it must be ensured that the "oop" handouts will not be distributed accompanying the "se" slides or vice versa. The simplest way of assuring the consistency between slides and corresponding handouts would be to combine both into an instance labeled "oop" or "se". Thus, in an attempt to avoid role instances we have just re-invented role instances.

This point about multiplicities is crucial because a number of researches still follows the dogma that role instances as a special case of "object schizophrenia" should simply be avoided. Such avoidance would, however, mean that certain situations simply cannot be modeled, except by the introduction of even more objects (which was to be avoided in the first place).

Dynamic view

Fields of a role are visible only when looking at the role, not at its base. Different concepts exist for how a role may at run-time access the shared state of its base. For illustration please confer to Figure 1: Role Participant abstractly defines two properties ***`name`*** and ***`matr`*** (for matriculation number). Both properties are not stored in the role but refer to corresponding fields in the underlying base of type Student.

In those approaches where role and base form a close union (let's called it the one-instance approach) also base fields are accessed as if they where direct fields of the role (i.e., the abstract declarations in Participant would not be needed). In other approaches some forwarding may be needed which would possibly involve one or more methods encapsulating the field. Forwarding may happen either explicitly or implicitly, the latter approach being able to simulate the one-instance approach on top of a model where role and base are distinct instances.

If field access is handled using message forwarding the dynamic view of fields will be subsumed in the dynamic view of methods.

2.3. Methods

Since methods and fields together define the properties (features) of an object, it should not surprise to see some commonalities between these two. However, while fields need to be allocated for each instance, methods have no such requirement. On the other hand methods are involved in control flows which adds to the complexity of the matters at hand.

Static view

Much like fields a role may share methods of its base object (as defined in the base object's class).

If *fields* of the same name exist in a role and its base at run-time multiple slots will exist. Instead of such replication, *methods* with multiple definitions may override each other. As a motivating example consider a method `answerPhone` in class Professor. Certainly the regular behavior of answering his (mobile) phone is not tolerable while the professor is giving a lecture. For this situation the role Lecturer should override the normal behavior with a behavior that is more suitable to this context.

A language combining object-oriented programming with roles may need to specify two different kinds of overriding: (1) overriding of a shared (inherited) method by a sub-class and (2) overriding of a shared base method by a role. Both kinds of overriding can be declared implicitly (based on name equality) or explicitly by some extra declaration.

The semantics of any kind of overriding requires a look at the runtime behavior, i.e., the dynamic view.

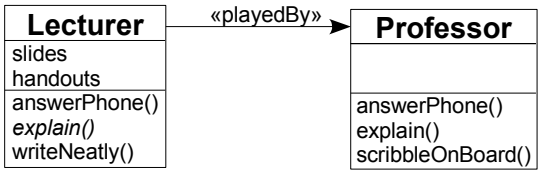

Fig. 3. Several methods in role and base.

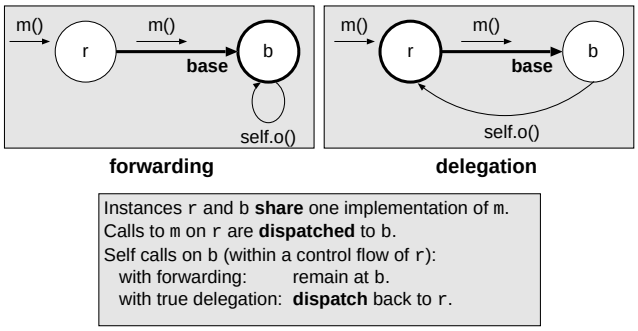

Fig. 4. Forwarding vs. Delegation.

Dynamic view

Dynamic method dispatch is maybe the central runtime mechanism of object-oriented programming languages. Role objects introduce some more options to method dispatch.[1] For illustration see Figure 3. The two versions of method `answerPhone` have already been mentioned: while the Professor is in class his normal behavior – even if triggered without knowledge about the Course – should be overridden. The Lecturer's method `explain` may not need a specific implementation as the normal behavior of Professor can be re-used. The realization of `explain` may internally use another method `scribbleOnBoard`, which in the Professor's office results in hardly legible scribblings, which only his PhD students can decipher. In order to make the method `explain` useful for the Course context, internal calls to `scribbleOnBoard` must use the method `writeNeatly` given in Lecturer (which produces neat writing but consumes more time). These scenarios can be mapped to different techniques of method dispatch.

Most commonly, sharing the behavior of its base is implemented as forwarding method calls from a role object to its base. Such dispatch may or may not apply true delegation with late binding of self.[2] The weaker form of *forwarding* (see left-hand side of Fig. 4) forgets about the initial call target (`r`, the role), and subsequent self-calls will remain at the base object `b`. Thus, forwarding does not allow methods of the parent object (here: `b`, the base object) to be overridden by methods of the child (here: role). For the example this would imply that forwarding `explain`-calls from the Lecturer to the Professor would result in illegible scribblings on the Course's black board. Note, that a pattern based implementation of roles in a standard object-oriented language Bäumer et al. (2000) can only support weak forwarding. Anything beyond weak forwarding requires explicit support by the language.

The stronger form of *delegation* (see right-hand side of Fig. 4) is used to define object-based inheritance which has basically the same power as ordinary class-base inheritance (see Stein et al. (1989) and our discussion of related work). The key idea behind true delegation is that the original call target (`r`) remains active even while executing `b.m()`, such that `self` within `b.m()` will in that case still be interpreted as referring to `r`. This interpretation of `self` allows code within `b` to actually invoke `r.o()`, which means, `o()` is effectively overridden. Using true delegation the

[1] Kappel et al. (1998) identify significant differences among proposed role languages regarding method dispatch.

[2] Please note, that the issue of forwarding versus delegation is of some general relevance in object-oriented programming. Yet, *any* concept of roles must provide an answer to this question.

Lecturer can actually combine the Professor's capability to explain with the Lecturer's capability of neatly writing to the board. True delegation brings the power of the Template-and-Hook pattern to the relationship between a role and its player, as can be seen when identifying `explain` as a template method and `scribbleOnBoard` as the contained hook.

A third concept may also be used for method dispatch: *method call interception*. By this technique, the control flow can be redirected from the original call target to some other instance. Redirecting a call to a base method as to invoke a role method instead has mainly the same effect as overriding in the context of object-based inheritance. It might, however, be interesting to distinguish whether or not dynamic dispatch is influenced by who initially triggered a behavior: Should a Professor completely abandon his original behavior simply because he is a Lecturer? When writing to a board, the adapted behavior should only be used when called (indirectly) from a Lecturer role. On the other hand, when the Professor's phone rings while he gives a lecture he should not care who is calling (regarding both phone call and method call).

Unfortunately, each of these three concepts, forwarding, delegation, interception, is in some publications called "delegation". In the context of design patterns the word "delegation" usually refers to the concept of forwarding. In the role-based approach of Albano et al. (1993) a concept of dynamically extending a role with a sub-role is defined, which uses "delegation" to dispatch to the lately added sub-role. This kind of dispatch should actually be classified as method call interception.

Such issues of dispatching are fundamental when designing programming language support for roles. For more abstract models it might be less important, but it is still important to note that a role may exhibit behavior that is similar to that of its base but with certain well-defined differences concerning additional methods and methods that are overridden in the role. More generally spoken, it must be defined how a base and its role(s) collaborate to perform the desired behavior.

2.4. Context

At the conceptual level, roles are frequently defined as named places in a relationship or the like (see, e.g., Steimann (2000)). In other words, roles define the intersection of objects and contexts, such that different contexts select different roles. Our example (Fig. 1) has introduced the contexts of different Courses and a PhD context.

Static view

A context groups a set of roles assigning a specific place to each of its roles. Such grouping creates a containment relation between the context and its contained roles.

Generally spoken it is a central property of roles to be defined only in relation to something. Defining explicit contexts provides an excellent means for separation of concerns. Within a given context roles may define *views* of actual entities in the world. It is the power of this concept to support focusing on *relevant* aspects only while ignoring other aspects.[3] This general requirement can be realized by explicitly defining the interfaces of roles in a way that certain features of the base are visible at the role (by way of forwarding or delegation) while other base features are not accessible via the role.

Referring back to Figure 1 the interface of course Participants has been designed as to provide access to the base's `name` and `matr` properties. All other properties of Students (like, e.g., the date of birth) are not visible properties of a Participant.

While of course many different views on Students are possible it is the *context* that discriminates into relevant and irrelevant aspects; relevance must always be seen relative to a given context.

[3] In this section the word "aspect" should be interpreted in its natural meaning. The related work section will discuss the relation between roles and the technical term "aspects" as in aspect-oriented programming.

In other words, the base object defines *what* we are looking at, while the context defines *why* we are looking at this object. After answering both questions, the role defines *how* exactly we are looking at the base object from the context's perspective.

Having explicit support for context mandates its use for a variety of reasons, some of which refer to existing design patterns:

- A context may model a relationship where the roles are the places in this relationship.
- A context may be a Mediator coordinating the collaboration of its roles.
- A context may be a Façade providing an interface for a set of roles which are not visible at the outside.
- A context may be a module for a scenario.

Despite these general observations, programming languages supporting roles commonly ignore the issue of context.

Dynamic view

Usually a role cannot migrate from one context to another, because it is the context that defines the role. However, entering and leaving a context is an indispensable concept with regard to roles.

The effect of entering a context can be described as some kind of activation. Only roles of a currently active context will ever be relevant for the execution of a program. There may be different ways of entering or activating a context.

A context can either be seen as a purely abstract entity that merely groups role-classes. Other approaches define context as first-class citizens, i.e., contexts can be instantiated resulting in identifyable instances at run-time. Note, that Figure 2 already uses context instances to distinguish different courses.

3. Roles in ObjectTeams/Java

In this section we will present how roles are realized in the programming language ObjectTeams/Java Herrmann (2002b)[4]. The language combines concepts from different research areas. It is mostly discussed in the context of aspect-oriented programming. Also collaboration-based design has strongly influenced the language design. Collaborations are modeled using a new kind of class-like module called *team*, hence the name Object Teams. ObjectTeams/Java (OT/J for short) is the incarnation of this programming model for the host language Java.

The example in Figure 1 is in fact a valid Object Teams model. It will be used as a starting point for illustrating details of OT/J. The presentation of roles in OT/J will follow the two by four system of coordinates given above. While we walk through the matrix we will show how the properties of roles captured in the matrix are mapped to constructs of OT/J.

We invite the interested reader to relate to the comprehensive language definition OTJLD (short for ObjectTeams/Java Language Definition) Herrmann et al. (2007).[5] The OTJLD contains the precise definition of all constructs mentioned in this paper. For easier reference we add links to the specific paragraphs of the OTJLD (e.g., an overview of the purposes of that document can be found in (OTJLD§0)).

General introductions to OT/J can be found in Herrmann (2002b); Veit & Herrmann (2003). Specific concepts and mechanisms are explained more in depth in Herrmann (2004a,b); Herrmann et al. (2005). The focus of the following sections lies on assessing the suitability of OT/J for implementing role-based models.

[4] For up-to-date software and documentation see `http://www.objectteams.org`.

[5] An up-to-date hypertext version of this document is always available at `http://www.objectteams.org/def`.

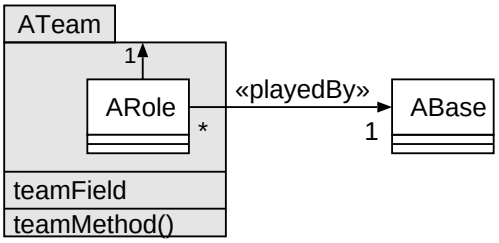

Fig. 5. Basic structure of role, team and base.

3.1. Role Instances

The primary decision in the design of the programming language ObjectTeams/Java is to model roles by specific classes and to define a `playedBy` relationship at the class level which specifies that each instance of the bound role class will be constantly attached to a given base instance (OTJLD§2.1):

```
class Lecturer playedBy Professor  ...
```

When illustrating Object Teams using the UML (see AOSD'02 (2002)) we draw an association with stereotype `«playedBy»` (see Fig. 5).

Static view

A `playedBy` declaration by itself results only in an invariant that is enforced for all instances of the role class: each instance will at creation time be attached to a base object of the declared type. This link is immutable throughout the life-cycle of the role object. This restriction is due to the fact that a base object may contribute essential properties to its role. In this setting an unattached role would break the type system resulting in an object that lacks some features which its type promises. In order to enforce this invariant the role–base link is completely hidden. By this regime no client code may ever temper with the base link. However, a number of language features, to be introduced below, is implicitly founded on the base link.

Statically, roles in ObjectTeams/Java allow any pattern of multiplicity discussed above. When attaching multiple roles of the same type to the same base instance, these roles must reside in different contexts in order to remain distinguishable.

In ObjectTeams/Java we furthermore distinguish between *unbound* and *bound* roles, where only *bound* roles, which have a `playedBy` clause, strictly conform to the prevailing notion of roles. An unbound role will *never* be attached to a base object. Still there is a number of properties that bound and unbound roles share (OTJLD§1.2). A bound role may inherit from an unbound role. Along role inheritance the `playedBy`-clause can be refined to a more specific base class (OTJLD§2.1(*c*)).

Dynamic view

We have already mentioned that bound roles in ObjectTeams/Java cannot be unattached during any time of their lifecycle. We have argued before why we think that this restriction is well tolerable. The decision to realize the role-base link by an *immutable* reference excludes the option of role migration, where a role could change its attachment from one base object to another.[6]

[6]The primary reason behind this decision was quite pragmatic: the concept of *immutability* already exists in Java (using the modifier `final`). Using the immutability property it was relatively easy to guarantee non-nullity for the base link, thus ensuring that every instance of a bound role class at any time has a valid reference to a base instance. Without immutability we would have to explicitly enforce a *non-null* property by our type system. In a language where potentially every variable may hold a null reference at run-time, non-nullity cannot efficiently be guaranteed for a few variables since none of the possibly-null variables could ever be used in the right-hand side of an assignment to a non-null variable. Once support for null-checking is well established in object-oriented programming, we should reconsider the decision in order to support role migration.

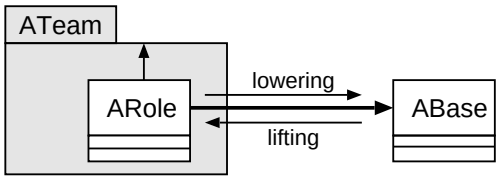

Fig. 6. Two directions of navigation/translation.

Looking however from the base object, full dynamism is granted as roles may be attached and removed from the base at any time. After unattached bound roles are ruled out, removing a role obviously also means to destroy it. These operations are supported by means of predefined reflective methods.

Navigation from a role to its base is trivial as mentioned before. However, ObjectTeams/Java adds a novel mechanism for navigating the opposite direction: from a base object to its role. This mechanism is called *lifting* (OTJLD§2.3).

The idea behind lifting is as follows: Whenever a base instance enters the context of a team instance (a context for roles, see section "Context" below), it is translated by the lifting mechanism to the corresponding role. In terms of the running example, whenever a Professor enters a class room, he or she will *become* a Lecturer, given that this is the only role a Professor can play in a Course. Similarly, when a Student is seen as a member of a Course, he or she is lifted to a Participant role.

As an example for lifting consider the question how to retrieve the grade for a Student "jane" with respect to a Course "se". Obviously neither of the given objects "jane" nor "se" has a field `grade`. The most elegant way to implement the query is by a method in type Course which takes the Student as an argument:

```
int getGrade(Student as Participant part) { return part.grade; }
```

Here the signature of `getGrade` uses *declared lifting* (OTJLD§2.3.2) to do the essential look-up: callers of this method must provide an instance of type Student, however, the method body receives an argument of type Participant. The translation is requested by the `as` keyword. It is transparently managed by the language, i.e., programmers never need to worry how to actually find an appropriate role for a given base instance. After this lifting translation the field access `part.grade` is indeed legal and safe.

Technically, each team maintains a mapping from base objects to role objects, such that a role of a given type is identified by the pair of a base and a team object. This mapping is well-defined due to the restriction mentioned above: it is not allowed for two roles of the same type and attached to the same base object to reside in the same team. Formally, lifting is a function

$$teamInst \times baseInst \times roleType \rightarrow roleInst$$

As can be seen from this signature, lifting mandatorily requires a team instance, thus lifting is not meant for requesting *all* roles of a given base object. The context must always be known for accessing a role which resides in that context.[7]

Opposite, whenever a role object shall leave the context of its enclosing team, it is translated by *lowering* to the base object it is attached to. Lowering is the inverse of lifting. It is implemented by simply navigating the base link (OTJLD§2.2). As an illustration for how the language manages lowering translations consider the following method of type Course:

```
Professor getProfessor() { return this.held_by; }
```

[7]The signature shows that lifting requires two instances: a team and a base. Therefor it would have been possible to allocate the mapping at the base objects instead of the team. However, base classes are developed and compiled without any knowledge of being bound to a role, whereas a team knows about its roles. Thus, it is far easier to let the team manage this mapping.

The return statement tries to pass a value of type Lecturer, whereas a return type Professor is specified. The language treats this mismatch roughly equivalent to an up-cast: each Lecturer is considered compatible to the type Professor, which is at run-time satisfied by translating the Lecturer to its underlying Professor instance using lowering.

Fig. 6 abstractly illustrates both kinds of translations.

The above syntax for declared lifting (using the `as` keyword) is in fact an exception from the general rule that lifting and lowering are handled *transparently*. The language is designed in such a way, that the compiler can determine which points in the execution of a program cause a data-flow across a team boundary (OTJLD§2.3(b), §2.2(b)). These data-flows apply lifting and lowering as needed to adapt types, with the highly desirable effect that the team context can be fully implemented in terms of its roles only, whereas the world outside a team should usually not explicitly refer to any role.

For reasons of similarities to subtype-polymorphism the concept of translations using lifting/lowering is called *translation polymorphism* Herrmann (2004b). It actually defines a new kind of substitutability based on wrapping/unwrapping objects. As mentioned, lowering resembles the situation of up-casts, which are always safe. Lifting on the other hand is similar to a down-cast, however, *without* the danger of failure (like throwing a `ClassCastException` in Java), because lifting will create the desired role instance if none already exists.

3.2. Fields

A role class may declare fields just like regular classes. Namespaces of a role and its base are disjoint.

Static view

Fields declared in a role class are of course allocated for each role instance. This avoids the problems of objects needing to grow dynamically, as we discussed in Section 2.2. Each new role instance naturally allocates the corresponding fields as needed.

Base fields are only shared if specifically declared. The exact construct for accessing base fields will be shown next.

Dynamic view

In order to access a field of its base object, a role must declare an indirection (OTJLD§3.5), as the following declaration in role Participant illustrates:

```
String name() → get String name;
```

Such a declaration in a bound role class grants read access to a field called `name` which must be defined in the base class (here: Student). The above declaration implicitly defines a role method `name()`. Using the modifier `set` instead `get` and declaring an appropriate method signature will in analogy grant write access. Any such field access from a role to its base need not pay attention to access restriction as defined in the base class, marking the fact that the role-base relationship is more intimate than normal associations (OTJLD§3.5(e), §3.4).

Forwarding a field access from a role to its base is a special case of so-called `callout` bindings to be defined below.

3.3. Methods

A role class may declare methods just like regular classes. Again, namespaces of a role and its base are disjoint.

Static view

A role may selectively share methods from its base and it may also override base methods. Any base method that is not explicitly shared is not visible for the role. Understanding the mechanisms behind sharing and overriding requires a look into the dynamic method dispatch.

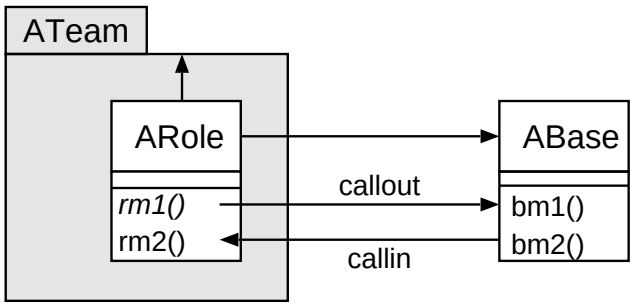

Fig. 7. Two kinds of method binding.

Dynamic view

When a role class declares to share a method of its base class this technically boils down to method *forwarding* along the `base` link. Such forwarding is established by a declarative method binding (OTJLD§3) (cf. Fig. 3):

```
explain → explain;
```

Object Teams features method bindings in two directions. The forwarding from a role to its base is called a *callout* binding. Just like in the `playedBy` clause, names of the role always appear at the left-hand side whereas names of the base are at the right-hand side.

Callout bindings may have different levels of detail. In any case the base method may be made available be a new name. Signatures may need to be given, and parameters can even be mapped (OTJLD§3.2) in order to mend little incompatibilities between the base-implementation and how this is to be seen in the team context.

Forwarding as established by a callout binding can be seen as lowering the call target. Similarly, method arguments may also need to be lowered during forwarding caused by a callout binding (OTJLD§3.3).

The point in declaring callout bindings is to establish a selective object-based inheritance relation. Each feature mapped in a callout binding is shared between base and role. Unmapped features are invisible at the role.

The careful reader might ask, whether a callout binding establishes simple *forwarding* or *delegation* with late binding of `this`, as it would be required for true object-based inheritance. The answer is that a callout binding by itself indeed causes simple forwarding. This has the effect, that during the execution of the base method no information is available that the method is being executed on behalf of a role. Therefor, any self-call within this method will normally stay at the base object. Object-based inheritance, on the other hand, allows the role to override methods of its base.

In ObjectTeams/Java forwarding and overriding are decoupled. It is the second binding direction by which method overriding can be specified. A *callin* method binding (OTJLD§4)

```
writeNeatly ← replace scribbleOnBoard;
```

has the effect, that calls to `scribbleOnBoard` will be *intercepted* and redirected to the role, invoking its own method `writeNeatly`[8]. The redirection caused by a callin binding involves lifting of the call target (to look up the base object's role) and also of method arguments if needed.

In all object-oriented languages, if a method is overridden in the context of regular inheritance, it is always possible to invoke the original (inherited) method using a super-call. The same applies to callin replacements in OT/J where a call to the original method is a achieved by a so-called "base-call", which forwards back to the original call target, the base instance. Instead of `super.m()` we simply write `base.m()`. Note that the symmetry of lifting and lowering hides

[8] Following the tradition of aspect-oriented programming also `before` and `after` modes are available

the fact that a message is being dispatched between different instances. Role and base behave as two sides of the same coin.

Summarizing the role-base relation, it can be described as a tunable object-based inheritance relation. It supports sharing (callout bindings), overriding (callin bindings) and substitutability (translation polymorphism).

3.4. Teams as Context for Roles

A central contribution of ObjectTeams/Java is the capability to group a number of roles into what we call a *team*. Teams combine the properties of classes and packages: Conceptually they are classes in that they have fields and methods, may apply inheritance[9] and are instantiable. At the same time, teams are packages in that they contain a set of classes. While this class containment is very similar to inner classes in Java, we also provide the option to physically store the inner classes in a directory representing the team (OTJLD§1.2.5), which emphasizes the package-face of teams. By this combination teams are better scalable than classes (even with inner classes) and they are first-class entities, which normal packages are not.

Visibility of methods and fields may use a team as a namespace to control encapsulation: private features are visible in their enclosing class only, protected features can be accessed by a team and all its roles, only public features can be seen from outside the team (OTJLD§1.2.1(*e*)).[10]

A team is declared as a class with the modifier `team`:

```
public team class Course ...
```

Any inner class of a team is automatically (i.e., without an additional modifier) a role class. In diagrams the two-fold nature of teams as package and class is illustrated by using a package symbol with additional compartments for fields and methods plus optionally a decorating "T" icon.

Static view

Teams introduce another invariant (this property is directly adopted from Java inner classes): each role instance is contained in a team instance. The role refers to the enclosing team instance by a non-null immutable link. Being an object, too, a team instance may also hold explicit links to contained role objects.

ObjectTeams/Java even provides means to enforce a strict discipline of encapsulation, which prevents that a *confined* role will ever escape the context of its enclosing team instance (OTJLD§7) Herrmann (2004a).

Various styles how modeling can benefit from having explicit support for context have already been discussed in Section 2.4.

Dynamic view

The discussion of lifting has focused on data flows across a team boundary: when entering the context of a team, a base object is lifted to its corresponding role, when leaving this context it is lowered to its base. We will also take a look on how teams influence control flows in the system. In order to do so, we must first define the concept of dynamic *team activation*.

At each point in time each team instance is either active or inactive. In our running example each instance of team Course would be activated for the time of each lecture, only. This has the effect that only for these periods the Professor changes his behavior `answerPhone`. As soon as the class is over the Course will be deactivated resulting in the normal behavior to apply again.

[9] Team inheritance applies the concept of *family polymorphism* Ernst (2001) to allow for type safe covariant refinement of a set of role classes. This concept is essential in the design of OT/J but it falls beyond the subject of this paper.

[10] Visibility deviates from the rules of inner classes in Java. In Java private members of a inner class are visible for the enclosing class, too. However, when regarding a team as a package, it must be possible to protect features of a role against access even by the containing module.

Activating a team object is defined as enabling all callin-bindings that are defined in any of its roles. Invoking a base method for which a callin binding exists while an appropriate team object is active has the effect of intercepting the method call and redirecting the control flow to a role within the active team.[11] Conversely, deactivating a team instance disables all its callin bindings, i.e., none of its roles will intercept any control flows. Team activation realizes the concept of entering a context in the sense that only roles of an active team will influence the behavior of the system.

As we already discussed also data flows are effected by entering/leaving a team context: passing a base object into a team causes the base to be translated to (wrapped with) an appropriate role, passing a role object out of a team causes it to be lowered. If a base object is lifted for which no role object exists at that point in time a role is implicitly created on-demand.

This process can be fine-tuned by so-called *guard predicates* (OTJLD§5.4) Herrmann et al. (2005). One possible use of a guard predicate is illustrated by the following code fragment taken from Herrmann et al. (2005):

```
1  public team class ATM {
2    Bank bank;
3    ...
4    // This role applies to all accounts that
5    // are not from the same bank as the ATM.
6    public class ForeignAccount
7      playedBy Account
8      base when (!(bank.equals(base.getBank())
9    {
10     debitWithFee  <- replace debit;
11     displayError  <- replace getBalance;
12     ...
13   }
14 }
```

The idea in this example is that an ATM treats "foreign" accounts differently from those belonging to the same bank as the ATM. The difference in behavior is modeled in the role ForeignAccount which overrides the methods `debit` (to charge an extra fee) and `getBalance` (to disallow displaying the current balance). Obviously, this role should not apply to *all* accounts. This is declaratively ensured by the guard predicate shown in line 8. Base objects for which this predicate evaluates to false will never be adapted with a ForeignAccount role. Essentially, the guard is evaluated whenever either of the methods `debit` or `getBalance` is called on a base object. In this situation the callin binding attempts to lift the base instance to a suitable role instance. This lifting is, however, blocked by the guard. As a result of this blocking, the callin binding does not fire for domestic accounts, their behavior is not adapted by a role.

Herrmann et al. (2005) also demonstrates that guards can be used at four levels of granularity (team, role, role method, callin binding) and that they can apply either *before* the lifting (base guard) or *after* (role guard). Details of these mechanisms are, however, beyond the scope of this paper.

Coming back to the fundamental issue of activation, two styles of activating a team can be distinguished:

– Whenever the control flow enters a team due to a call to a team-level method, this team object is implicitly active until the method terminates (OTJLD§5.3).

[11] If more than one such team is active simultaneously, priority is determined mainly by the order of activation (OTJLD§5.1, §4.8).

– Additional means exist to imperatively activate any number of team objects (OTJLD§5.2).

The policy of *implicit* team activation ensures that callout-defined forwarding only happens while the team containing the role is active. By this rule all callin bindings of the role are effective while the base method is executing. The desired behavior of overriding is established.

Imperative team activation adds another option. Even when a method of a base object is invoked by a client which is *unaware of any roles* there may be a role in a currently active team which may influence the behavior of the base object. Deactivating the team restores the original behavior. This policy goes beyond traditional approaches to object-based inheritance. Those approaches apply overriding only when a method is explicitly invoked on a role instance. In our experience it is very desirable to be able to express that a role by its mere existence changes the behavior of the base to which it is attached, even for clients which are unaware of the role. Team activation and callin bindings can easily achieve this.

The concepts of callin bindings, team activation, guard predicates and lifting span a rich design space. The common theme behind this group of concepts is the explicit formulation of situational dependence of certain behaviors. Encoding such situational dependence in a more traditional programming language requires much greater efforts in terms of inserting complex conditional logic into many scattered places of the program.

3.5. On Assessment

Practical applications show that ObjectTeams/Java is indeed useful for building real world software systems. However, the problem in assessing a programming language lies in the word "useful".

During several case studies we have collected data about code sizes, about structural properties of the code as well as about the development process regarding productivity and maintainability. All these data support our approach, yet they can never cover one central issue: ObjectTeams/Java explicitly makes use of terms like "team", "role" et cetera, in order to appeal to the developers' intuition. This is in contrast with many recent programming languages, which, e.g., apply technical terms as "inter-type declarations", "mixins" etc.

The advantage of more "natural" terms as opposed to more "technical" terms can of course not be measured, yet the subjective feedback we received from developers indicates that the concepts of "roles" and "teams" actually inspire developers towards adequate and good software designs. We understand that this is well in line with the experience that an ontology based on the notion of roles also helps to build better domain models.

3.6. Postlude: why identity is a non-issue

Technically, roles in ObjectTeams/Java are distinguishable instances. The `==` operator will answer false when comparing a role and its base. Conceptually, we still think of a role–base pair as one entity. The automatic translation by lifting/lowering effectively supports this conceptual unity. In day-to-day programming with ObjectTeams/Java we have not yet come to a situation, where it was actually relevant to check for the identity of a role and its base, and where this check should indeed answer true. However, the easiest of all solutions to the seeming dilemma (see 2.1) has already been given by Richardson & Schwarz (1991): we only need to define two separate operators: `==` will continue to distinguish a role from its base whereas a second operator (in Richardson & Schwarz (1991) `@=` is used) considers all roles as identical to their base. In ObjectTeams/Java we refrained from adding a new operator but simply prefer a predefined method `roleEQ`.

Of course lifting and lowering greatly help to avoid the comparison of identity of roles and bases: within the team everything can be expressed using roles only, outside the team roles should be avoided alltogether. It could in fact be considered bad style if one wanted to compare a role object to a base object.

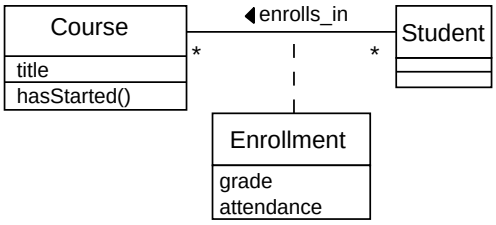

Fig. 8. An example for a relationship with an association class.

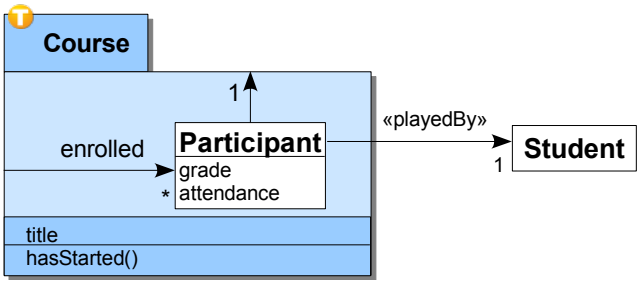

Fig. 9. Modeling a relationship using a team.

4. Teams vs. Relationships

Object Teams focuses on embedding roles into a context called "team". Another concept that is closely related to roles is "relationship". One may argue that also relationships deserve direct and first-class representations in a programming language. When regarding relationships as communication protocols, roles in Object Teams directly correspond to roles in a protocol. In this case the enclosing team is nothing but a module around the pieces defining the protocol.

The correlation is a little more difficult to see, when regarding relationships from a data modeling perspective as in E/R modeling. To illustrate correlation in this case, we start with a textbook example of a relationship as shown in Figure 8. In E/R modeling a primary reason for making relationships first-class is the modeling power that results from admitting relationships to have attributes just like classes. When moving from traditional E/R modeling to the UML this is translated to associations having an association class. The standard example (Fig. 8) reads as follows: The `enrolls_in` relationship between students and courses is enriched by the association class `Enrollment` which allows to capture additional properties for each pair in the $Course \times Student$ relationship. If only an object-oriented language is used for implementing this model, one has to revert to introducing some kind of manager object that explicitly maps a Student to his or her records regarding a specific Course. This auxiliary structure is needed for each instance of Course.

We have already seen a corresponding model in Object Teams, which only cosmetically differs from the model in Fig. 8: The association class Enrollment is renamed to Participant and it is made a role contained in a Course. The original `enrolls_in` relationship is split into its two directions: a regular directed association `enrolled` leads from a Course to its Participants, from where the Students are reachable using lowering. Reversely, a lifting relationship allows to navigate from a Student to its Participant role. Note, that the lifting relationship can already be seen as a ternary relationship, relating an instance of each of Student, Course and Participant. Such a ternary relationship is exactly how the three classes in Fig. 8 are connected.

The magic behind our solution lies in the language support for the lifting relation: each team implicitly implements that functionality that in a plain object-oriented solution would have to be realized by an explicit manager class. Each team instance *is* already a manager for all its contained roles. The required book-keeping is given for free, just by using the keywords `team` and `playedBy`.

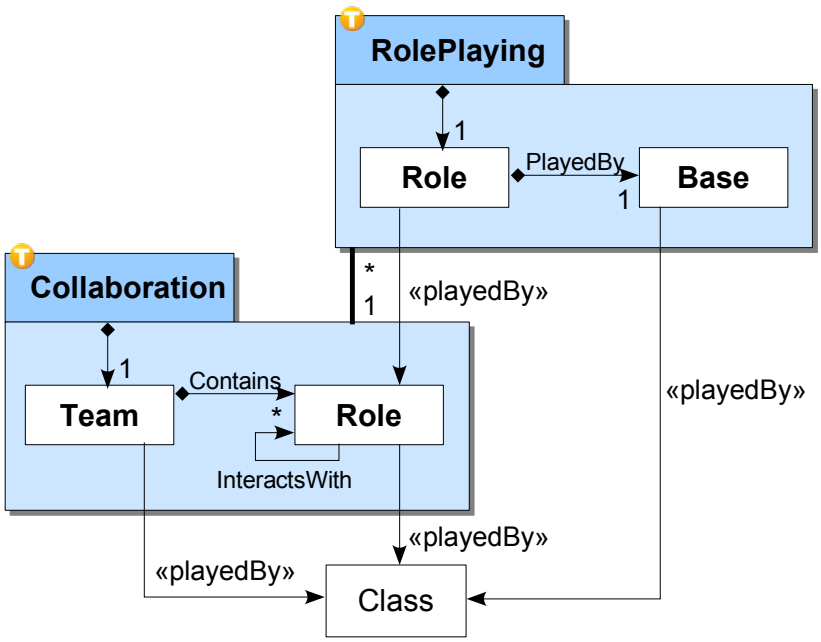

Fig. 10. A core meta model for Object Teams.

5. A Meta Model for ObjectTeams/Java

In this section we present a core meta model for ObjectTeams/Java in order to summarize the structural relationships and outline some options how these concepts can be combined in actual models.

During the development of several tools for ObjectTeams/Java we have developed several meta models that on the one hand were technically motivated and on the other hand applied standard object-oriented constructs for meta modeling as a way to facilitate an implementation in Java. While doing so we fell into the trap of a naive meta model which contains three fundamental classes: Team, Role and Base. While this looks natural at first, this approach fails to supports a number of constellations that are in fact essential to Object Teams.

Those constellations have been identified previously as nesting, stacking and layering Herrmann (2003). Nesting, e.g., refers to the idea that a team can be contained (nested) within an outer team. However, by virtue of being a member of the outer team, the nested team is automatically a role, too. This leads to a model, where Teams, Roles and Bases are not disjoint sets, but actually each intersection is populated with legal elements, too.

Since overlapping classes are problematic in object-oriented design, we choose to use Object Teams as the language for our meta model. The resulting core meta model can be seen in Figure 10. The central idea behind this model is to leave the meta class Class untouched, i.e., we refrain from sub-classing Class to produce Team, Role and Base. Instead we identify additional properties that can be attached to a Class if it appears in a certain *context*. Classes appearing in a Collaboration may play either the role of the Collaboration's Team, or they may play a role of a Role within the Collaboration. This models the fact that Teams and Roles *only* occur in the context of a Collaboration, where each Collaboration has exactly one defining Team and any number of interacting Roles.

It is easy now to represent team nesting as the example in Figure 11 illustrates. At the bottom part of this Figure you see a concrete model depicting a team TOuter, which contains two roles, R1 and TR. The latter is also a team containing an inner role R2. Above the dotted line you see named instances of the meta model representing exactly the model from below. The interesting element in this picture is TR. Basically, it is an instance of Class, but it also appears in two

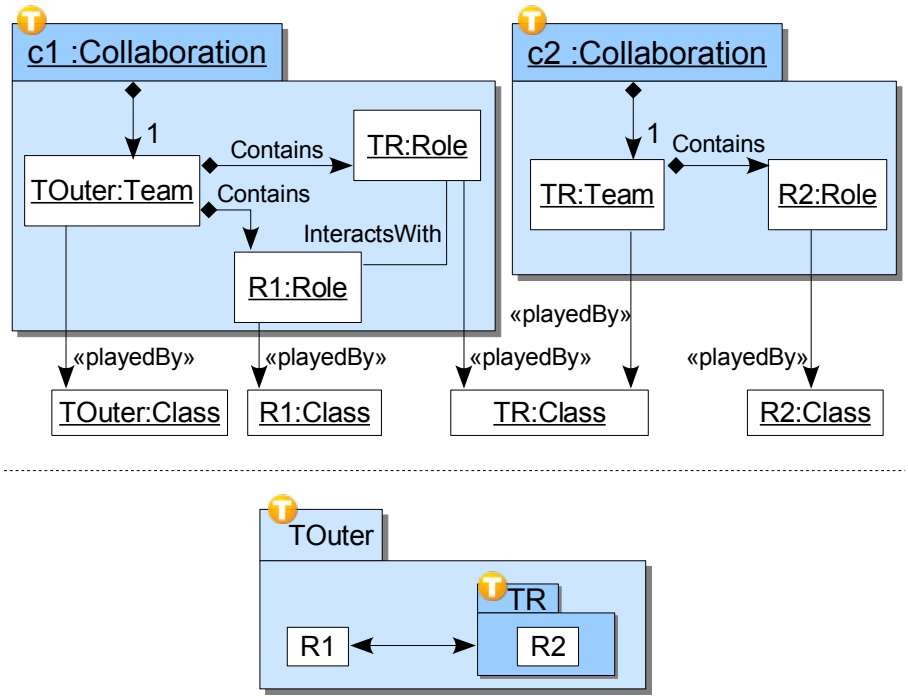

Fig. 11. Team Nesting: concrete model and its representation using meta objects.

different contexts: Within collaboration c1 it plays the role TR:Role interacting with role R1, whereas in the other it plays the role TR:Team containing role R2. Thus, the conceptual entity TR consists of three parts which in conjunction define that TR has the properties of a Class *plus* the properties of a Role (with respect to collaboration c1) *and* of a Team (with respect to collaboration c2). This constellation could not be modeled easily using classes only.

In a similar vein a Class could have both a Base role and a Team role (which we call team stacking), and a Class could have both a Base role and a Role role (which we call team layering) Herrmann (2003).

Another issue that can be captured nicely in this meta model is the following dependence: if a RolePlaying context should be created, which defines one class as the Role and another class as the Base of a PlayedBy relationship, the RolePlaying depends on the existence of a Collaboration and more specifically, the Role of RolePlaying depends on the fact that the corresponding Class already is a Role in a Collaboration. This dependence is expressed by modeling that RolePlaying.Role is played by Collaboration.Role. By transitive role playing it is ensured that the top role can only exist if both the lower role and its base exist.

Thus, the meta model itself is an example of "team layering", which stands for a situation where the roles of an upper team are played by roles of a lower team, each team adding a layer of structure and behavior to the system beneath. In our meta model team RolePlaying defines a layer on top of team Collaboration. Both parts are joined in the class Collaboration.Role, which is both a role of Collaboration *and* the base of RolePlaying.Role.

To further clarify terminology we identify a Collaboration.Role that has no RolePlaying.Role associated to it as an *unbound role* whereas a RolePlaying.Role is always a *bound role*.

6. Applying Steimann's criteria

In this section we iterate through the list of features given in Steimann (2000). For each item we will briefly demonstrate how it is supported in ObjectTeams/Java.

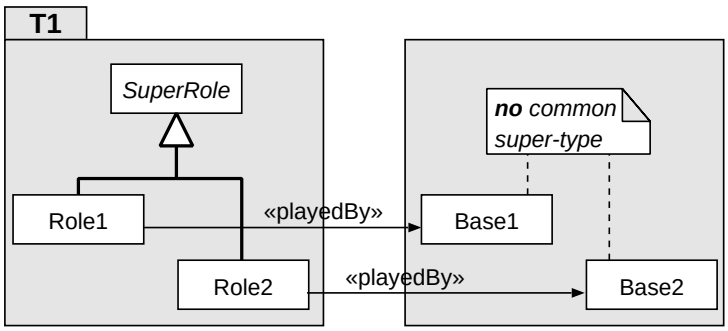

Fig. 12. Mapping one role to several unrelated base classes.

(1) *A role comes with its own properties and behavior.* Role classes may declare fields and define methods.

(2) *Roles depend on relationships.* In the previous section we have shown how relationships with association classes can be translated into Object Teams models. Also, a team can be used to model an n-ary relationship with all its roles being the "named places" that link to the underlying base objects connected by the relationship. Since roles actually depend on an enclosing team this feature is supported by ObjectTeams/Java.

(3) *An object may play different roles simultaneously.* A base object may have roles in different teams. It may also have several different roles within the same team, which is only subject to certain structural restrictions which are needed to keep the type-system sound (OTJLD§2.3.4).

(4) *An object may play the same role several times, simultaneously.* For the same base object and the same role type, multiple roles can exist if they live in different team instances, which makes them distinguishable. Thus, the only restriction in multiplicities is that OT/J does not allow multiple roles of the same base object that are not distinguishable, neither by role-type nor by team instance (cf. the signature of lifting on page 10). We could find no natural example (i.e., not involving computers), were the same player plays the same role more than once within the same context.

(5) *An object may acquire and abandon roles dynamically.* Being separate instances, it is no problem to create/destroy roles independently of their base. Roles may come into being either by explicit creation or by an implicit lifting translation. Lifting can be further controlled by guard predicates.

(6) *The sequence in which roles may be acquired and relinquished can be subject to restrictions.* Interestingly, Steimann's example Steimann (2000) to this item illustrates a dependency of a role requiring another role as its base (see item (8)). While roles-of-roles can indeed express static requirements at the type level, role guards in OT/J can additionally impose arbitrary dynamic constraints on the automatic instantiation of a role. Callin bindings can be used to trigger the acquisition of a role.

(7) *Objects of unrelated types can play the same role.* Although not supported by a specific language construct, this features falls off easily using an unbound role and several sub-classes (roles) which are bound to different base classes (see Fig. 12). In this example `Base1` and `Base2` — although being unrelated types — may both have a role of type `SuperRole` (realized either as `Role1` or `Role2`). This pattern allows all roles to share some implementation from the unbound role and also adaptations to different base classes can be achieved by different callout mappings in the sub-classes. Another interpretation of this pattern is, that the unbound role introduces an a-posteriori super-type over (views of) a set of base classes.

(8) *Roles can play roles.* Since roles are instantiable, it is not a problem to attach one role to another. Typing such constructs is, however, not trivial. In ObjectTeams/Java a role and its base may not be contained in the same team instance. Practical use of ObjectTeams/Java has shown that indeed very nice layered structures can be realized using roles-of-roles (see also Sect. 5).

(9) *A role can be transferred from one object to another.* To-date, ObjectTeams/Java does not support this feature. Type-safety will never allow a role instance, whose class declares a `playedBy` relation, to exist without a valid `base` link. Currently this constraint is enforced by the immutability of the `base` link. If, more generally, non-null analysis would be used instead (which is not practical in the current type-system of Java), it would be easy to add an API-function that migrates a role from one base to another.

(10) *The state of an object can be role-specific.* Indeed, being an instance, each role may carry specific state.

(11) *Features of an object can be role-specific.* Fields and methods can be defined in role classes just like in any class. In Steimann (2000) the explanation of this item mentions multiple overloading of the same feature. ObjectTeams/Java supports overloading, renaming and overriding.

(12) *Roles restrict access.* All base features not bound by a callout binding remain invisible.

(13) *Different roles may share structure and behavior.* Here Steimann lists the options that (a) a role class may inherit features from a super role and (b) a role instance may share features from its base using delegation. In OT/J (a) is given by regular inheritance between role classes. (b) is controlled by callout bindings, which selectively expose base features to the role.

(14) *An object and its roles share identity.* Using `roleEQ` comparison, a role and its base appear as having one shared identity. Translation polymorphism establishes substitutability between a role and its base.

(15) *An object and its roles have different identities.* Using the `==` operator the identity of roles can indeed be distinguished from each other and from their base.

6.1. Evaluation

From applying the 15 features we conclude that ObjectTeams/Java almost fully supports all desirable properties of roles. Only role migration (9) cannot easily be reconciled with static type-safety in a language without proper null-analysis.

Some features might not be supported by specific declarative constructs, but rather need a little explicit programming. Most notably, item 6 (sequence) could of course be supported by a much more specific sub-language such as regular expressions. However, having guard predicates in the language is already a great help for this task and we would consider more specific support for this item as undue language bloat. ObjectTeams/Java even supports features which were said to be contradictory (14 and 15) by passing the option to the programmer whether a specific identity check should be strict (`==`) or not (`roleEQ`).

7. Discussion and Related work

When discussing the concept of roles in the context of ontologies, it is a common understanding that roles depend on a player and/or a context Masolo et al. (2004); Loebe (2007). This corresponds to the structural constraints imposed on roles in OT/J, where unbound roles depend on the enclosing team, only, whereas bound roles additionally depend on a player. Unlike Loebe (2007) we do not introduce a player universal, i.e., we regard the extension of players as an implicit subset of the base class's extension rather than making this extension explicit. Loebe (2007) also distinguishes different kinds ("types") of roles (relational, processual, social). By contrast, OT/J only provides a very generic concept for roles, which should be suitable for modeling all kinds of roles. This means, while the concept of roles *is* made an intrinsic part of the language, further distinctions are left to concrete models written in the language. In comparison to work on foundations of ontologies, the design of OT/J intentionally restricts the number of structural concepts rather than emphasizing the number of variations and perhaps corner cases that are observed, e.g., in sociology.

On the other hand, a programming language *must* be complete and precise with respect to behavioral aspects, which are not discussed in this depth in works on ontologies. We hope that the considerations about forwarding, delegation, overriding and dispatch in general help in understanding the behavioral relationships between roles, players and contexts.

Steimann (2000) discusses the tension whether roles are super-types or sub-types of their "natural types" (base types). He concludes that two separate hierarchies should be maintained for natural types and role types that are connected by a role-filler relationship. This separation into unrelated concepts (roles and natural types) significantly reduces the possibilities to combine the fundamental concepts, e.g., roles of roles are not possible in his approach. OT/J accounts for the tension between roles as sub-types and super-types by the concept of translation polymorphism. The answer in OT/J is: strictly speaking no sub-typing relation exists between a role and its base in either direction. However, *substitutability* is given in *both* directions using lifting and lowering. Additionally, generalizing over unrelated base types is supported by the indirection shown in Figure 12.

Programming language support for roles can be traced back all the way to SELF Ungar & Smith (1987) which demonstrated the universal applicability of object-based inheritance paving the road for the Treaty of Orlando Stein et al. (1989). OT/J supports both kinds of sharing discussed in Stein et al. (1989): (1) Inheritance enables a sub-class to share all features of its super-class; inherited methods can be overridden; inheritance can be applied to all kinds of classes including teams and roles. (2) The playedBy relation together with callout bindings enable a role to share selected features of its base class; by callin bindings a role class can override any method existing in the base class.

Aspects as introduced in Richardson & Schwarz (1991) already come quite close to our understanding of roles.[12] This is one of the earliest approaches that indeed integrates a concept of roles ("aspects"), which is based on delegation, into a class-based and statically typed language. Interestingly, already aspects have the ability to rename a feature that an aspect acquires from its base. This corresponds to our concept of callout bindings, which may also use different names at the base and role sides. We are not aware of an implementation of the Aspects concept. A comparison of five approaches including Richardson & Schwarz (1991) can be found in Kappel et al. (1998). In this paper a special focus lies on analyzing the different strategies of method dispatch, which supports our understanding that indeed method dispatch is a key feature for the integration of a role concept into a programming language. None of the approaches discussed in Kappel et al. (1998) support a concept of context.

For the development of ObjectTeams/Java, Aspectual Components Lieberherr et al. (1999) were a major source of inspiration. In that model contexts are called "components" containing a number of "participants". A separate "connector" is used for binding participants to base classes. A connector can define method bindings, however, the distinction of binding directions (callout vs. callin) is not explicit in that approach.

In the tradition of Aspectual Components, Dynamic View Connectors Herrmann & Mezini (2000) uses the techniques for component integration and the language prototype Lua Aspect Components (LAC Herrmann & Mezini (2001)) directly paved the road for ObjectTeams/Java. More recently, CaesarJ Mezini & Ostermann (2003) can also be seen as a successor of Aspectual Components and also shows some features which have been introduced in ObjectTeams/Java. CaesarJ, however, leaves more of the infrastructure code to be written explicitly by the programmer. E.g., the base link is a plain reference to be used directly in client programs, and lifting (called "wrapper-recycling") must be requested explicitly. The difference can be interpreted such that CaesarJ defines a more abstract model with the goal to capture more styles of programming, whereas OT/J more directly relates to the metaphors of roles and teams.

[12] This notion of "Aspect" is independent of and pre-dates "aspect-oriented programming".

On a different branch, Truyen (2004) has developed similar technology called LasagneJ for the purpose of supporting multiple client-specific contexts in distributed systems. The motivation behind LasagneJ is to support context-sensitive customization of services that are shared over the internet. While LasagneJ supports instantiable collaborations similar to our teams, a collaboration in LasagneJ is instantiated only once per application and statically associated to a feature identifier. This supports the desired customization but it does not make collaborations first-class entities as teams are in OT/J.

Finally, the Chameleon model Graversen (2004) follows the tradition of Kristensen & Østerbye (1996). Chameleon might support a few more corner cases regarding individual roles, but in that work support for context has not been fully elaborated.

Several of the approaches mentioned above are also influenced by aspect-oriented programming (AOP) Kiczales et al. (1997). The connection is twofold: (1) Roles and aspects are two different incarnations of the concept to use *views* to partition a model, knowing that this kind of partitioning has to deal with overlap. AOP is negatively defined as a solution to the problems of crosscutting (scattering and tangling), i.e., the inability to modularize several concerns simultaneously if using traditional object technology only. Due to this negative definition the (technical) notion of aspects has no commonly accepted intrinsic meaning. As a consequence the use of aspects for conceptual modeling is controversially debated, to say the least. To the contrary, roles have initially been defined for the sake of conceptual modeling. After precise semantics have been added to the concept, roles have a very good coverage of all phases in the software life-cycle. (2) Both roles and aspects add a fundamentally new kind of control flows to programming languages, that transcends the explicit style of method calls. Roles apply *method call interception* to allow for a role instance to override methods of its base instance. Aspects more generally apply join point interception to weave aspect behavior into a base application. The difference between roles and aspects is the set of available join points: roles normally override whole methods, only. Aspects on the other hand can hook into individual instructions of some kind like: calling a method, throwing an exception or accessing a field (depending on the language's join point model). The common theme is, that parts of a program ("base") can be adapted without modifying the base's source code. We claim that method call interception in the vein of overriding is far easier to understand than complex "pointcuts" referring to sets of individual instructions. Also maintenance, and more specifically refactoring, is significantly more difficult in an aspect-oriented program than in a role-based program. Some more ideas on this comparison can be found in the contributions to a workshop called "Views, Aspects and Roles (VAR'05)"[13]

Aspect-oriented programming is also used in Pearce & Noble (2006) for implementing relationships as first-class entities by so-called Relationship Aspect Patterns. The patterns presented mainly solve the problem that bi-directional relationships tend to be scattered over both classes involved when using standard object-oriented techniques. The use of AOP is in fact limited to so-called intertype declarations, which are a means to add features to a class without editing the class's source code. The same designs can be realized using roles instead of aspects, where the features to be added are implemented as features of a role. Pearce & Noble (2006) also touches some performance issues. When transforming the design to use roles, instantiation and look-up will cause some overhead. However, with just a little help the OT/J compiler could transparently inline those role-objects which would then produce roughly the same performance properties as the solutions in Pearce & Noble (2006). On the other hand, Relationship Aspect Patterns address only issues of data modeling and thus support a very limited notion of roles, actually only the two generic roles "from" and "to" of a binary relationship are mentioned.

More recent approaches which also combine roles with explicit context are powerJava Baldoni et al. (2005) and EpsilonJ Tamai et al. (2005). Despite being developed independently these two languages exhibit strong similarities among each other and with OT/J. Most of the concepts of

[13] See `http://swt.cs.tu-berlin.de/~stephan/VAR05`

powerJava and EpsilonJ can easily be mapped to corresponding concepts in OT/J demonstrating that all three languages are converging towards a common solution. In powerJava and EpsilonJ the issues of role attachment and the base-to-role navigation are handled explicitly in the source code. Most of this is subsumed in our concept of lifting. Through the mostly implicit nature of lifting we achieve a better separation between the worlds of roles and base objects. Both mentioned languages lack the concept of persistently activating a context.

In this paper we have related ObjectTeams/Java to the general concept of roles as classified in Steimann (2000). Our analysis has shown, that ObjectTeams/Java fulfills most of the requirements for roles, to an extent that, to the best of our knowledge, has not been demonstrated for any other approach, yet.

7.1. Current state and future work

The design of the language ObjectTeams/Java is mostly complete and is well integrated with all features of its host language Java, as documented in the comprehensive language definition Herrmann et al. (2007). As a final step in language design we have designed a sub-language for selecting *join points* for aspect integration. Note, that callin bindings already provide a basic mechanism for aspect-oriented programming in ObjectTeams/Java. The join point language has not yet been integrated into the stable branch of the tooling. Following the arguements above, even when sophisticated join point is available, this technique should be used only sparingly, as it may conflict with the demands of maintenance and especially refactoring.

The compiler for ObjectTeams/Java is based on the Eclipse Java Development Tooling. Eclipse has been extended in many ways to effectively support the development of software using ObjectTeams/Java. This IDE is freely available[14] and we have performed an industrial case study for empirically evaluating the benefit of using ObjectTeams/Java for software development. This case-study plus class-room use and diploma theses give rise to confidence in ObjectTeams/Java being a sound approach that actually supports improved modularity and a very good alignment from conceptual modeling to an implementation using the very same concepts.

At a technical level applying OT/J for the implementation of its own tools Herrmann & Mosconi (2007) has demonstrated significantly improved maintainability of components that re-use and adapt existing components.

Design notations for Object Teams have been discussed Herrmann (2002a); Herrmann et al. (2004) and prototypical tool support using UML profiles and code generation is available.

In our research we mainly follow a bottom-up approach. We hope that providing a programming language, whose technical details have been settled over a number of years, provides a solid foundation for developing a comprehensive method for seamless software development using not only objects but also roles and collaborations as first-class concepts throughout the software lifecycle. Defining a programming language that precisely defines the high-level abstractions “team” and “role” proves that these concepts are indeed well-defined. We hope that ontology-driven approaches can be aligned with the concepts presented in this paper, such that an implementation in OT/J could be the final result in a seamless development process. Most of the success of object-orientation is due to an improved seamlessness over earlier paradigms. We are convinced that Object Teams have the potential of taking seamlessness one step further and that software developed in the new approach may exhibit better modularity and also traceability from requirements to code.

[14] `http://www.objectteams.org/distrib/otdt.html`.

Acknowledgment

We would like to thank an anonymous reviewer who has suggested to provide a meta model of Object Teams *using* Object Teams, an idea that actually solved a dilemma of previous meta models we had used.

References

Albano, A., Bergamini, R., Ghelli, G. & Orsini, R. (1993). An object data model with roles. In *VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases* (pp. 39–51). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

AOSD'02 (2002). *Proc. of First International Conference on Aspect Oriented Software Development*. Enschede, Netherlands: ACM Press.

AOSD'03 (2003). *Proc. of 2nd International Conference on Aspect Oriented Software Development*. Boston, USA: ACM Press.

Baldoni, M., Boella, G. & van der Torre, L. (2005). Roles as a coordination construct: introducing powerjava. In *Proceedings of MTCoord05*.

Bäumer, D., Riehle, D., Siberski, W. & Wulf, M. (2000). *Pattern Languages of Program Design 4*, chapter Role Object, (pp. 15–32). Addison-Wesley.

Ernst, E. (2001). Family polymorphism. In *Proc. of ECOOP'01*, number 2072 in LNCS (pp. 303–326). Springer Verlag.

Graversen, K. (2004). Role collaborations. In *Workshop on Software-Engineering Properties of Languages for Aspect Technologies (SPLAT) at AOSD'04*.

Herrmann, S. (2002a). Composable designs with UFA. In *Workshop on Aspect-Oriented Modeling with UML at AOSD'02 (2002)*.

Herrmann, S. (2002b). Object Teams: Improving modularity for crosscutting collaborations. In Aksit, M., Mezini, M. & Unland, R. (Eds.), *Proc. Net Object Days 2002*, Volume 2591 of *Lecture Notes in Computer Science*. Springer.

Herrmann, S. (2003). Orthogonality in language design – why and how to fake it. In *Workshop on Object-oriented Language Engineering for the Post-Java Era, at ECOOP*. Darmstadt.

Herrmann, S. (2004a). Confinement and representation encapsulation in object teams. Technical Report 2004/06, Technical University Berlin.

Herrmann, S. (2004b). Translation polymorphism in Object Teams. Technical Report 2004/05, Technical University Berlin.

Herrmann, S., Hundt, C. & Mehner, K. (2004). Mapping use case level aspects to objectteams/java. In *Workshop on Early Aspects at OOPSLA'04*.

Herrmann, S., Hundt, C., Mehner, K. & Wloka, J. (2005). Using guard predicates for generalized control of aspect instantiation and activation. In *Dynamic Aspects Workshop (DAW'05), at AOSD 2005*. Chicago.

Herrmann, S., Hundt, C. & Mosconi, M. (2007). ObjectTeams/Java Language Definition version 1.0 (OTJLD). Technical Report 2007/03, Technische Universität Berlin.

Herrmann, S. & Mezini, M. (2000). PIROL: A case study for multidimensional separation of concerns in software engineering environments. In *Proc. of OOPSLA 2000*.

Herrmann, S. & Mezini, M. (2001). Combining composition styles in the evolvable language LAC. In *Proc. of ASoC workshop at the 23nd ICSE*.

Herrmann, S. & Mosconi, M. (2007). Integrating object teams and OSGi: Joint efforts for superior modularity. In *Proc. of TOOLS Europe 2007, also in: Journal of Object Technology, 6(9), 2007*.

Kappel, G., Retschitzegger, W. & Schwinger, W. (1998). A comparison of role mechanisms in object-oriented modeling. In K. Pohl, A. Schürr, G. V. (Ed.), *Proceedings Modellierung'98* (pp. 105–109).

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. & Irwin, J. (1997). Aspect Oriented Programming. In *Proceedings of ECOOP '97*, number 1241 in LNCS (pp. 220–243).

Kristensen, B. & Østerbye, K. (1996). Roles: Conceptual abstraction theory and practical language issues. *Theory and Practice of Object Systems*, *2(3)*, 143–160.

Lieberherr, K., Lorenz, D. & Mezini, M. (1999). Programming with Aspectual Components. In *Technical Report*. Northeastern University.

Loebe, F. (2007). Abstract vs. social roles – towards a general theoretical account of roles. *Applied Ontology, In this issue*.

Masolo, C., Vieu, L., Bottazzi, E., Catenacci, C., Ferrario, R., Gangemi, A. & Guarino, N. (2004). Social roles and their descriptions. In *D. Dubois, C. Welty, M.A. Williams (eds.), Proceedings of the Ninth International Conference on the Principles of Knowledge Representation and Reasoning (KR2004)* (pp. pp. 267–277). Whistler, Canada.

Mezini, M. & Ostermann, K. (2003). Conquering aspects with caesar. In AOSD'03 (2003).

Pearce, D. J. & Noble, J. (2006). Relationship aspect patterns. In *Proc. of the European Conference on Pattern Languages of Programs (EuroPLOP)*.

Richardson, J. & Schwarz, P. (1991). Aspects: extending objects to support multiple, independent roles. In *Proceedings of the 1991 ACM SIGMOD international conference on management of data*.

Steimann, F. (2000). On the representation of roles in object-oriented and conceptual modelling. *Data and Knowledge Engineering*.

Steimann, F. (2001). Role = interface: A merger of concepts. *Journal of Object-Oriented Programming*, *14(4)*, 23–32.

Stein, L. A., Lieberman, H. & Ungar, D. (1989). A shared view of sharing: The Treaty of Orlando. In W. Kim & F. H. Lochovsky (Eds.), *Object-Oriented Concepts, Databases and Applications* (pp. 31–48). Reading (MA), USA: ACM Press/Addison-Wesley.

Tamai, T., Ubayashi, N. & Ichiyama, R. (2005). An adaptive object model with dynamic role binding. In *Proc. International Conference on Software Engineering (ICSE2005)* (pp. 166–175).

Truyen, E. (2004). *Dynamic and context-sensitive composition in distributed systems*. PhD thesis, Katholieke Universiteit Leuven.

Ungar, D. & Smith, R. B. (1987). Self: The power of simplicity. In *Proc. of OOPSLA'87*.

VanHilst, M. & Notkin, D. (1996). Using role components to implement collaboration-based design. In *Proc. of OOPSLA'96*, Volume 28(10) of *ACM SIGPLAN Notices*.

Veit, M. & Herrmann, S. (2003). Model-View-Controller and Object Teams: A Perfect Match of Paradigms. In AOSD'03 (2003).

Wieringa, R. & de Jonge, W. (1995). Object identifiers, keys, and surrogates. *Theory and Practice of Object Systems*, *1(2)*, 101–114.