

Programming with Roles in ObjectTeams/Java

Stephan Herrmann
Technische Universität Berlin
stephan@cs.tu-berlin.de

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 focusses on some aspects of roles while neglecting others. We present the programming language ObjectTeams/Java and using the categories of Steimann[17] 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.

1. INTRODUCTION

The concept of roles appears in a variety of research areas. While the concept is widely accepted for conceptual modeling, some other areas are more reluctant. This paper does not argue for the benefit of using roles. Nor does it try to define what roles are or give a comprehensive comparison of existing approaches using roles. We will simply present a programming language, ObjectTeams/Java, and relate this language to previous work on roles.

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 ObjectTeams/Java along those coordinates (Sect. 3). When all relevant features of the language have been presented we will apply the criteria defined by Steimann[17] to ObjectTeams/Java, concluding a nearly complete coverage of all desirable properties (Sect. 4). We conclude with a discussion of related work (Sect. 5).

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 also 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.

2. SYSTEM OF COORDINATES FOR ROLES

Steimann [17] has presented a list of 15 features by which roles have been characterized in different approaches. This paper will investigate the concept of roles along a two-by-

four matrix, which can be seen as abstractions over Steimann's 15 features.

On the one axis of our system of coordinates we distinguish a **static** and a **dynamic** view. Along the other axis we will discuss instances, fields, methods and context.

2.1 Instances

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 the larger part of this paper we will simply assume that roles indeed have their own identity, because for some desirable properties of roles we see no viable solution without role instances. We will, however, come back to this issue in Sect. 3.5 and 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. 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.

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. A role may be attached to a base object at any time during the base object's life-cycle. The same holds for removing a role from its base. In some approaches it is even possible to keep unattached roles and change the attachment by moving a role from one base object to another.

In this context we should also look at how the relation between roles and bases is used for navigation between objects. It is trivial to allow for navigation from a role to its base but approaches may or may not provide means for navigating from a base object to one of its roles.

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 state of a role is usually defined by the union of its direct fields and the fields of its base object. 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. 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. Selecting between the different values for the same field would require some additional unique key. Since such keys are definitely needed, we see no benefit in denying that these keys actually correspond to an identity of each role instance.¹

This point about multiplicities is crucial because legions of researchers still follow the dogma that role instances as a special case of "object schizophrenia" should simply be avoided. For a large number of models such avoidance is a blank ostrich approach.

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 access the shared state of its base. In those approaches where role and base form a close union also base fields are accessed as if they were direct fields of the role. In other approaches some forwarding may be needed which would possibly involve one or more methods encapsulating the field.

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, methods need no instantiation and on the other hand control flow comes into focus here.

Static view. Much like fields a role may share methods of its base object (as defined in the base object's class). Instead of replication methods with multiple definitions may override each other, the semantics of which requires a look at the runtime behavior, i.e., the dynamic view.

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.

Most commonly, sharing the behavior of its base is implemented as forwarding method calls from a role object to its

¹One could still argue that the mentioned keys would be scoped to the respective base object without the need to be globally unique, but we see no fundamental gain in such distinction.

base. Such dispatch may or may not apply true delegation with late binding of self. This stronger form of delegation is used to define object-based inheritance which has basically the same power as ordinary class-base inheritance. By contrast, the weaker forwarding forgets about the initial call target (the role) and does not allow methods of the parent object (here: base object) to be overridden by methods of the child (here: role).

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.

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.

2.4 Context

At the conceptual level, roles are frequently defined as places in a relation or the like. In other words, roles define the intersection of objects and contexts, such that different contexts select different roles.

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.

While some approaches consider this a central property of roles to be defined only in relation to something, programming languages supporting roles commonly ignore this issue.

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 for 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.

3. ROLES IN OBJECTTEAMS/JAVA

In this section we will present how roles are realized in the programming language ObjectTeams/Java[8, 15]. The language combines concepts from different research areas. It is mostly discussed in the context of Aspect Oriented Programming. Also collaboration-base 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 is the incarnation of this programming model for the host language Java.

The presentation of roles in ObjectTeams/Java will follow the two by four system of coordinates given above.

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:

```
class MyRole playedBy MyBase ...
```

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.

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. A bound role may inherit from an unbound role. Along role inheritance the `playedBy`-clause can be refined to a more specific base class.

Dynamic view.

We have already discussed the restriction regarding the immutable base link. Looking however from the base object, full dynamism is granted as roles may be attached and removed from the base any time. Removing a role obviously also means to destroy it.

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*.

The idea behind lifting is as follows: Whenever a base instance enters the context of a team instance (a role context, see Sect. 3.4 below), it is translated by the lifting mechanism to the corresponding role. Technically, each team maintains a mapping from base objects to role objects, such that a role is identified by the pair of a base and a team object. Opposite, whenever a role object shall leave the context of its enclosing team, it is translated by *lowering* (the inverse of lifting, implemented by simply navigating the base link) to the base object it is attached to.

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. These data-flows apply lifting and lowering as needed to adapt types, such that the team context can be implemented fully in terms of its roles, whereas the world outside a team should usually not explicitly refer to any role.

For reasons of similarities to subtype-polymorphism the concept of these translations using lifting/lowering is called *translation polymorphism* [11]. It actually defines a new kind of substitutability based on wrapping/unwrapping objects.

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. 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, as in:

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

Such a declaration in a bound role class grants read access to a field called `name` which is assumed to be defined in the base class using an implicitly generated role method `getName`. 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.

Forwarding a field access from a role to its base is a special case of so-called `callout` bindings to be defined in Sect. 3.3.

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.

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:

```
requiredRoleMethod → existingBaseMethod
```

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 by a new name. Signatures may need to be given, and parameters can even be mapped 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.

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 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 `overridingRoleMethod ← replace someBaseMethod2` has the effect, that calls to `someBaseMethod` will be redirected to the role invoking its method `overridingRoleMethod`. The redirection caused by a callin binding involves lifting of the call target (to lookup the base object's role) and also of method arguments if needed.

Similar to method overriding with regular inheritance the previous version can be invoked, with **base** instead of **super**.

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 concepts of classes and packages: They are classes in that they have fields and methods, may apply inheritance and are instantiable. They 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 store the inner classes in a directory representing the team, which emphasizes the package-face of teams.

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

```
public team class MyTeam ...
```

Any inner class of a team is automatically (ie., without an additional modifier) a role class.

Static view. Teams introduce another invariant (this property is directly adopted from Java inner classes): each role

²Following the tradition of aspect-oriented programming also **before** and **after** modes are available

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. Rules for good design using Object Teams mandate the use of teams for a variety of reasons, some of which refer to existing design patterns:

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

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[10].

Dynamic view.

Two styles of entering and leaving the context of a team can be distinguished. (1) Whenever the control flow enters a team due to a call to a team-level method, this team object is active until the method terminates. (2) Additional means exist to imperatively activate any number of team objects.

The effect of activating a team object is that all callin-bindings defined in one of its roles are enabled. 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.³

Aside from its effect on control flows, 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 implicitly creates a role on-demand. This process can be fine-tuned by so-called *guard predicates*[4]. Guards can be defined at several locations within role or team classes. They can be used to filter which calls to base methods should actually be intercepted due to callin bindings. A guard can also be used to prevent on-demand creation of a role.

The above policy ensures that callout-defined forwarding only happens while the team containing the role is active. Therefor 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

³If more than one such team is active simultaneously, priority is determined mainly by the order of activation.

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 opinion 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.

3.5 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 [16]: we only need to define two separate operators: `==` will continue to distinguish a role from its base whereas a second operator (in [16] `@=` 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 altogether.

4. APPLYING STEIMANN'S CRITERIA

In this section we iterate through the list of features given in [17]. For each item we will briefly demonstrate how it is supported in ObjectTeams/Java.

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

(2) *Roles depend on relationships.* In a good design using ObjectTeams/Java, a team will model one or more relationships, with its roles being the places in the relationship. Roles indeed depend on their enclosing team instance.

(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.

(4) *An object may play the same role several times, simultaneously.* For the same base object and the same role type, roles can still be distinguished if they live in different team instances.

(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.

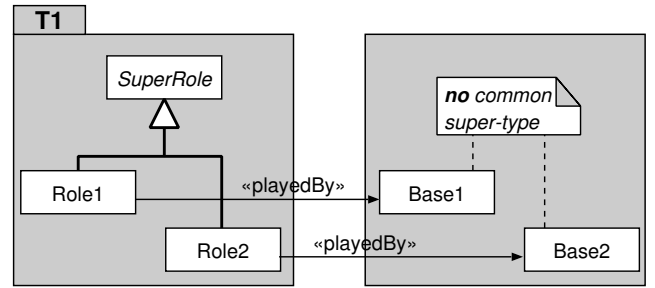


Figure 1: Mapping one role to several unrelated base classes.

(6) *The sequence in which roles may be acquired and relinquished can be subject to restrictions.* Interestingly, Steimann's example[17] to this item illustrates a dependency of a role requiring another role as its base (see item (8)). Role guards can in fact impose arbitrary 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 feature falls off easily using an unbound role and several sub-classes (roles) which are bound to different base classes (see Fig. 1). 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.

In [9] we have presented three general patterns of composing systems from roles and teams, like, e.g., team nesting, where a team is again a role of an even larger team.

(9) *A role can be transferred from one object to another.* To-date, ObjectTeams/Java does not support this feature. Typesafety will never allow a role instance, whose class declares a `playedBy` relation, to exist without a valid `base` link. Transferring a role from one base to another could easily be achieved by an additional API-function. We just haven't seen the need for this, yet.

(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. 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.* Features are selectively shared using callout bindings.

(14) *An object and its roles share identity.* Using the `roleEQ` 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.

4.1 Evaluation

From applying the 15 features we conclude that ObjectTeams/Java fully supports all desirable properties of roles. 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.

5. RELATED WORK

Programming language support for roles can be traced back all the way to SELF[20] which demonstrated the universal applicability of object-based inheritance paving the road for the Treaty of Orlando[18].

ASPECTS [16] already comes quite close to our understanding of roles. More recently, three branches of research can be identified, which put basic mechanisms for roles into a broader context.

For the development of ObjectTeams/Java, Aspectual Components [13] were a major source of inspiration. In this branch Dynamic View Connectors [5] uses the techniques for component integration and the language prototype Lua Aspect Components (LAC [6]) directly paved the road for ObjectTeams/Java. More recently, Caesar [14] can also be seen as a successor of Aspectual Components and also shows some features which have been introduced in ObjectTeams/Java. Caesar, 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.

On a different branch, LasagneJ [19] has developed similar technology for the purpose of supporting multiple client-specific contexts in distributed systems.

Finally, the Chameleon model [2] follows the tradition of [12].

Despite their different roots these three branches have come to very similar results. In all these approaches, roles (or wrappers) are grouped into larger modules, which seems to be the major advance over earlier models like [16]. However,

a systematical comparison among these branches has not yet been given.

In this paper we have related ObjectTeams/Java to the general concept of roles as classified in [17]. 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.

5.1 Current state and future work

Design of the language ObjectTeams/Java is mostly complete. A last step in language design will be the integration of 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 second-generation 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 ([15]) and we are currently performing an industrial case study for empirically evaluating the benefit of using ObjectTeams/Java for software development. Classroom use, diploma theses and early results from the case study 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.

Design notations for Object Teams have been discussed[7, 3] and tool support using UML profiles and code generation is under way. In our research we mainly follow a bottom-up approach. We hope that providing a programming language, whose technical details have been settled over the last 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 life-cycle. 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.

6. REFERENCES

- [1] *Proc. of First International Conference on Aspect Oriented Software Development*, Enschede, Netherlands, 2002. ACM Press.
- [2] Kasper Graversen. Role collaborations. In *Workshop on Software-Engineering Properties of Languages for Aspect Technologies (SPLAT) at AOSD'04*, 2004.
- [3] S. Herrmann, C. Hundt, and K. Mehner. Mapping use case level aspects to objectteams/java. In *Workshop on Early Aspects at OOPSLA'04*, 2004.
- [4] S. Herrmann, C. Hundt, K. Mehner, and J. Wloka. Using guard predicates for generalized control of aspect instantiation and activation. In *Dynamic Aspects Workshop (DAW'05) at AOSD'05*, 2005.

- [5] S. Herrmann and M. Mezini. PIROL: A case study for multidimensional separation of concerns in software engineering environments. In *Proc. of OOPSLA 2000*. ACM, 2000.
- [6] S. Herrmann and M. Mezini. Combining composition styles in the evolvable language LAC. In *Proc. of ASoC workshop at the 23rd ICSE*, 2001.
- [7] Stephan Herrmann. Composable designs with UFA. In *Workshop on Aspect-Oriented Modeling with UML at [1]*, 2002.
- [8] Stephan Herrmann. Object Teams: Improving modularity for crosscutting collaborations. In M. Aksit, M. Mezini, and R. Unland, editors, *Proc. Net Object Days 2002*, volume 2591 of *Lecture Notes in Computer Science*. Springer, 2002.
- [9] Stephan Herrmann. 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, 2003.
- [10] Stephan Herrmann. Confinement and representation encapsulation in object teams. Technical Report 2004/06, Technical University Berlin, 2004.
- [11] Stephan Herrmann. Translation polymorphism in Object Teams. Technical Report 2004/05, Technical University Berlin, 2004.
- [12] B. Kristensen and K. Østerbye. Roles: Conceptual abstraction theory and practical language issues. *Theory and Practice of Object Systems*, 2(3):143–160, 1996.
- [13] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components. In *Technical Report*, Northeastern University, April 1999.
- [14] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *Proc. AOSD'03*, Boston, USA, March 2003. ACM Press.
- [15] Object Teams home page. <http://www.ObjectTeams.org>.
- [16] J. Richardson and P. Schwarz. Aspects: extending objects to support multiple, independent roles. In *Proceedings of the 1991 ACM SIGMOD international conference on management of data*, 1991.
- [17] F. Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data and Knowledge Engineering*, 2000.
- [18] L. A. Stein, H. Lieberman, and D. Ungar. A shared view of sharing: The Treaty of Orlando. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 31–48. ACM Press/Addison-Wesley, Reading (MA), USA, 1989.
- [19] Eddy Truyen. *Dynamic and context-sensitive composition in distributed systems*. PhD thesis, Katholieke Universiteit Leuven, October 2004.
- [20] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proc. of OOPSLA'87*, 1987.