Why Most Domain Models are Aspect Free

Friedrich Steimann
Universität Hannover
Institut für Informationsysteme
Fachgebiet Wissenbasierte Systeme
Appelstraße 4, D-30167 Hannover
steimann@acm.org

DISCLAIMER

This is not a paper against aspects. In fact, I take my hat off to the people who have given us ASPECTJ and all the other excellent tools that have made AOP become reality and let us all find out for ourselves what aspects can do for us. This is not even a paper against aspect-oriented modelling. Quite the contrary: the way we model today (and presumably also the way we will model tomorrow) is inherently aspect-oriented, and the development of reliable weaving techniques is—in my view—at the core of MDA. It is however a paper against the belief that the aspects of AOP are modelling concepts that—on the same level as classes, attributes, and methods—are readily identified in every problem domain if only one looks at it with the right glasses on.

1. INTRODUCTION

Since the term AOP came public at ECOOP in 1997, workshops and conferences on aspect-related matters have literally mushroomed. Today we witness attempts to rewrite large parts—if not all—of software engineering to become aspect oriented: aspect-oriented design, aspect-oriented modelling, aspect-oriented requirements engineering, and so forth. One may ask oneself whether this enthusiasm is a sign of something revolutionary having been discovered, or just a symptom of the general pressure felt by the OO community to come up with something suitable to fill the hole called "post OO". Does aspect orientation really come with the substance necessary to found a new software development paradigm, or is it just another term to feed the old buzzword-permutation based research proposal generator?

That aspects can revolutionize software engineering analogous to the way objects did would require that aspects are an equally general notion, one that applies to the domains hosting computing problems as well as to the technology used to solve them. At first glance, this would seem case: when looking at a problem, we usually find that it has many aspects, that indeed every aspect comes with its own set of problems. We can even say that the objects of a domain themselves have different aspects, so that viewing aspects as a primitive concept of object-oriented software development would only seem natural.

Yet an aspect is immanently something observed of an object (or a problem), it is not itself one (or part of one). This is also reflected in natural language, where we usually speak of the aspects *of* something, not of the aspects *in* something. In fact, it seems that aspects reside one level above what is being looked at or, in other words, that aspects are a meta-level construct. Although aspects are not alone in this regard, I will argue below that this—together with a few other peculiarities—explains why we cannot expect to find aspects (at least not in the aspect-oriented sense) *in* any but a few rather special problem domains.

The remainder of this paper is organized as follows. First I identify different uses of the term aspect as relevant in the context of modelling. As I will argue, these uses are either better covered by other concepts or lie outside the subject of a model, i.e., do not refer directly to the modelled domain. Based on these findings I attempt a theoretical argumentation explaining why aspects (in the aspect-oriented sense) are necessarily second-order constructs and hence extrinsic to the problem domain and its models, which focus on the nature (the intrinsic properties) of the things being looked at. A discussion of my thesis with some of the relevant literature concludes my position.

2. DIFFERENT USES OF THE TERM *ASPECT* IN MODELLING

While technically the concept of an aspect is unambiguously defined by the aspect-oriented (modelling) language being used, conceptually it is not: people have different conceptions of what an aspect is and, consequently, of how and where it can be identified in a given subject matter. This is only natural since aspect is a general term in broad use not only in software engineering, but also in everyday conversation; like the term object before, it is readily adopted by everyone, but acceptance and popularity come at the price of precision.

What follows is a brief discussion of the different uses of the term *aspect* as used in software modelling. The discussion may be incomplete, yet I believe it covers the most important points being taken in the literature.

2.1 Aspects as Roles

Long before AOP, the database and the conceptual modelling community discovered that objects can have different *facets*, *perspectives*, *roles*, or *aspects* [14]. The classic example of a class whose instances have many roles[1] is `Person`: `Employee`, `Employer`, `Customer`, `Student`, and so forth. Many different ways to deal with roles have been proposed; most frequent are approaches that treat roles as subtypes, as supertypes, as a combination of both, or as adjunct instances [15]. All share the same least intent: to let an object have different properties in different contexts at different times.

There is however another important aspect to roles: objects of different types having same properties. For instance, many things in a modelled domain may be billable (play the role of a `Billable`), but these things need not be naturally related. On the programming side, we have roles such as `Serializable`, `Comparable`, `Printable`, etc., which are implemented by the most different classes. Technically, these are all *role types* allowing assignment compatible objects of otherwise arbitrary types to play the associated roles in the context of serialization, comparison, and printing, respectively. Conceptually, there is no difference between a document's being printable and a person's being employable; both require that the objects have certain properties that enable their functioning in the context defining the role. These properties are comprised in a corresponding role type.

Role types complement the natural partitioning of a problem domain (based on the natural types of objects, i.e., their classes) by one that is based on relationships and the contexts they produce. Given that roles partition a domain, one might argue that they crosscut it in the sense that they let several otherwise unrelated classes share same properties. However, although these properties are same, they are usually realized differently, reflecting the different nature of the objects possessing them—they are in fact polymorphic. Factoring out different implementations to a single place as suggested by an aspect-oriented approach would seem inapt, since it would contradict the most basic object-oriented principles.[2] Instead, interfaces (specifying protocol, but lacking implementation) and multiple (interface) inheritance readily lend themselves to representing roles and role playing, respectively, with mix-ins stepping in to allow for the inheritance of code wherever deemed appropriate [16, 17].

In object-oriented software modelling, roles are tied to collaborations: they specify what it takes for a single object to contribute to fulfilling some joint system functionality. Collaborations are based on interactions of objects; specification of such an interaction is typically not tied to a single role, but is distributed over all that contribute. Aspects on the other hand are typically defined orthogonally of one another; in fact, it is the very spirit of aspect orientation that aspects remain ignorant of each other. It follows immediately that modelling the roles of a system as aspects works only in cases where roles are isolated and monomorphic.[3]

All this in not to say that aspect technology has nothing to contribute to role modelling. In fact, role-oriented modelling (in the spirit of OORAM [13]) requires some kind of weaving, since it is not sufficient that the objects (of the classes) playing the roles of a collaboration guarantee to conform to the interface specification (or contract) associated with each role: the way the state of the same object playing different roles at the same time is to be shared or kept separate must also be specified. Because roles of *different* collaborations are defined largely independently of each other, some kind of weaving has to be performed when merging the different roles into the implementation of one class. However, given that every class implements its roles differently (the general case), it is difficult to conceive how aspect weaving mechanisms can help without major modifications. Aspectual collaborations [7] address these problems in some detail, but use roles in the specification of aspects, without equating the two concepts (*cf.* discussion in Section 4).

To summarize: a role is a named type specifying a cohesive set of properties whose specification is determined by the collaboration with other roles and whose implementation by different classes is typically polymorphic. Although conceptually a role of an object can be viewed as an aspect of it, this aspect is typically not one in the aspect-oriented sense.

2.2 Aspects as Ordering Dimensions

Ever since Aristotle, taxonomical orderings have been regarded as useful for structuring complex domains. However, the problem with taxonomies is that they can be based on different criteria, which may be independent of each

---

[1] In order not to confuse aspects and roles (which basically mean the same thing in this subsection, but do not in the remainder of this paper), we use the term *role* here.

[2] In fact, it would effect to reversing the *Replace Conditional with Polymorphism* refactoring [3]: code treating different objects differently would not be attached to the objects, but located in a single place, a conditional (typically a switch statement) branching on the type of the objects. Although aspects could be made polymorphic (*cf.* Section 4), this does not better the situation, since the definition of role-playing objects would remain scattered.

[3] One might argue that there are roles whose implementation is the same throughout, so that they are naturally represented by aspects. For instance, "having an address" (role *Addressee*) is something that applies to the most different objects, but has the same implementation everywhere. However, this does not preclude *Addressee* from being modelled as a role, particularly as this would allow its objects to participate in a *send* collaboration (with roles *Addresser* and *Addressee*), which the aspect does not. *Cf.* the discussion for more on this issue.

other. Different views (or aspects) on a domain may therefore lead to different orderings which, without one dominating the other, are difficult—if not impossible—to unify.

The introduction of polyhierachies (and multiple inheritance) combining several alternative classifications seems an immediate remedy. On closer inspection, however, they introduce more problems than they solve, since they tend to obscure the original orderings they are trying to combine—not without reason, major programming languages such as JAVA and SMALLTALK have abandoned the concept. The Unified Modeling Language UML [10] on the other hand has a special *discriminator* construct used to separate different dimensions ("partitionings") of a model's generalization/specialization hierarchies; however, as mere labelling this has no further-reaching effect on the structure of a model. In fact, keeping the dimensions separate (the aspect-oriented way) seems to be the best bet for maintaining accessibility of the domain. However, this does not mean that domains come with aspects, as the following reasoning shows.

The archetypal domain having conflicting ordering principles is the taxonomy of species. Its traditional version is based on externally visible properties such as number of legs, reproductive system, etc. Although the discovery of new species and even whole kingdoms requires reorganization from time to time, biologists have managed to keep the taxonomy in a strict tree form. Modern genetics however has made it possible to reconstruct the evolutionary development of the different species right from the first protists, thereby creating a taxonomy based on common ancestors rather than observables, entailing that it cannot be forced into strict tree form. While both *evolution* and *similarity* can be viewed as different *aspects* structuring the same problem domain, we observe that neither of these aspects is itself an element of the domain. Aspects as ordering principles describe the order, not the domain; hence, they reside one level above what they order.[4]

2.3 Domain-Specific Aspects

It has been noted many times that literally all aspects discussed in the literature are technical in nature: authentication, caching, distribution, logging, persistence, synchronization, transaction management, etc. One may add that these are all rather universal aspects, an observation that naturally begs the question whether all aspects are general, or whether there is such a thing as a domain-specific aspect.

A comparison with classes springs to mind: while we have general purpose, technical classes such as `String`, `Vector`, and `Exception` in a program, we usually also have domain-specific, non-technical classes such as `Account`, `Loan`, and `Currency`; in fact, the latter are the classes that are being modelled during the early phases of software development, since they represent the problem domain.

On closer inspection, it becomes clear that the standard aspects are aspects of programming rather than aspects of the domain the program is applied in: caching is a programming problem, as are logging, security, transaction management, etc.[5] In fact, we can observe that these aspects are aspects of the solution and its artefacts, not of the original problem. While this explains why the aspects are all technical (programming is a technical matter, and looking at it from different perspectives necessarily reveals its technical aspects), it also sheds a different light on domain specificity: an aspect is considered domain-specific if it occurs only in few, rather special programming problems. Note that the same domain specificity can be observed of classes: `Thread` for instance is specific to domains that exhibit concurrency, and it is technical (part of the solution, unlike for instance `PatientRecord`, which is a domain-specific, non-technical class).

Seen this way, we can expect to find new aspects while we address new problems (e.g., aspects of compiler construction, aspects of middleware, aspects of webs services, etc.), but these aspects will be domain-specific only in the sense that they address a programming problem that is specific to the domain—they are not themselves part of the domain. In fact, we can expect that every framework comes with its own set of aspects, and aspects will keep being discovered as long as technological advances are being made. But most—if not all—of these aspects will be specific to the technical solution (the "domain", if you will), not to the concrete problem it is applied to.

2.4 Aspects of Modelling

Now if the aspects we find when programming are *aspects of programming*, not of the programmed problem, then we may expect that the aspects we find when modelling are really *aspects of modelling*. And indeed, the aspects we can immediately identify are aspects of such kind: a static and a dynamic aspect, a component view, a use case view, etc. The fact that it has aspects is part of the nature of modelling, as it is part of the nature of programming; however,

---

[4] This argumentation also applies to other abstraction mechanisms such as classification and composition: an object can be classified according to its natural type (e.g., a `Person`, not a `Thing`) or to its technical type (e.g., an `Object`, not a `Class`); it can be a component of another object in the same problem domain, or of a deployment, etc. None of the ordering dimensions are themselves part of the ordered domain.

[5] Having said this, we note that sometimes a technical aspect has a parallel in the problem domain: in the perennial ATM example, for instance, transactions and logs are also entities that occur in the problem domain. However, these are in the same league as customers, accounts, and terminals: they are neither crosscutting nor do they exhibit other aspect-oriented peculiarities, so that they would preferably be implemented as ordinary types.

this provides no evidence that there are aspects *in* the domain being programmed or modelled, unless in very special cases (for example if the modelled domain is modelling itself).

Undoubtedly, modelling (much more than programming) requires some kind of weaving, since every model (model here defined as a single diagram) usually specifies only one tiny aspect of a modelled problem. In fact, I would conjecture that the weaving of diagrams (as partial models) is one of the key issues to be addressed if modelling is to deliver on its promises, MDA especially. I suspect that much can be learnt from AOP that can be extremely helpful in developing object-oriented modelling into a truly useful discipline, but I would expect none of this to relate to the level of the actual model, that is, to the conceptualization of the problem domain.

Given that roles have properties that make them unsuitable for being modelled as aspects, that the ordering function of aspects lifts them one level above the problem domain, and that the aspects we know of are really aspects of the solution and its technology rather than the underlying problem domain, are we ready to conclude that most domains are aspect free? No, since it could be the case that there are aspects I have forgotten to mention or that we do not even know of yet. What we really need is a line of reasoning making the claimed non-existence plausible or, better still, a proof of thereof.

3. REASONS FOR NON-EXISTENCE

There appears to be broad consensus in the conceptual, the data, and the software modelling community that the world be viewed as interrelated objects with attributes and behaviour. According to this view, objects are abstractions of real world entities (where we must be aware that even the concept of an entity is an invention of the mind), and their properties describe how entities appear, how they relate to others, and how they behave. While objects are the subjects of modelling, properties are "about" (or "above", which is the same word in German) them: not coincidentally, the most successful formalization of natural language, predicate logic, distinguishes between objects (zeroth-order expressions) and propositions about them (first-order expressions). As an aside, it is interesting to note that reality itself is usually free of propositions, unless of course "reality" (the modelled domain) is language.

3.1 The First-Orderedness of Domain Models

Being a picture of reality, a domain model consists of objects (representing the perceived entities of the real word) and propositions about them. In particular, a *pure domain model* contains no propositions about propositions, since these would describe the model rather than reality. As it turns out, first order predicate logic is the natural language of domain models, even in presence of object-orientation, i.e., typing, generalization, and inheritance. The following explains why this is so.

The standard semantics of object-oriented modelling maps the objects of a model to elements of the modelled domain. Types are mapped to unary predicates (called *type predicates*) serving as membership functions: an object o is an instance of type T iff $T(o)$ is true. Attributes correspond to functions associating certain elements (the objects) with others, their attribute values. Relationships between objects are mapped to binary or higher arity predicates, specifying tuples of elements that go together. Methods can be viewed as temporary relationships that objects engage in while collaborating; they introduce dynamics to a model in that they have the ability to alter existing relationships and attribute values as the result of their execution. [18]

The generalization of types expresses type inclusion, i.e., the fact that elements of one type are always (and necessarily) also elements of another type. More specifically, that T is a subtype of U maps to

$$\forall o : T(o) \rightarrow U(o) \qquad (1)$$

where o ranges over all objects in the domain and T and U are the corresponding type predicates. From this, the semantics of generalization, the inheritance of properties, follows immediately: whatever is asserted of objects of type U must also hold for objects of type T.

Because sentences of the form of Eq. 1 occur repeatedly in object-oriented models (they express the type hierarchy), one is led to view them as instances of a second-order relationship, one that relates types (and thus predicates) rather than objects. In fact, in a model we would not write Eq. 1, but

$$T < U \qquad (2)$$

or something alike. However, generalization as a second-order relationship is only *extensionally* defined (i.e., by listing all its elements)—it rolls out to a finite set of first-order formulas of the kind of Eq. 1. And indeed, even though Eq. 2 suggests that that type T inherits the properties from type U (matching the operational semantics of many popular programming languages), it is really the objects that inherit.[6] Inheritance is a proposition about objects and, thus, first order.

It is an interesting result of mathematical logic that that many-sorted (typed) and also order-sorted (object-oriented) logic are no more expressive than their uni-sorted forerunner: as long as they do not quantify over propositions, they are all first order, i.e., their sentences consist of objects (zeroth order) and propositions about them (first order) [9]. Thus, the fact that a model is object-oriented does not negate that it is a pure domain model in the above sense. As we will see, this is generally not the case for aspect-oriented models, which typically quantify over open (potentially infinite, in any case *intensionally* defined) sets of propositions.

[6] If anything, types inherit the declaration of properties.

3.2 The Second-Orderedness of Aspects

Frankly, the claim is that aspect-oriented languages are essentially second-order languages, so that their models are no pure domain models in the above sense. The second order follows from the fact that it is necessary for an aspect to be able to make propositions about propositions. In ASPECTJ, this is reflected in the fact that an aspect definition usually contains clauses specifying *where* the aspect applies, and this specification involves variables (wildcards and other constructs) ranging over classes, methods, and control flow. Mathematically, this is comparable to a second-order predicate logic in which variables may range not only over objects, but also over predicates and functors. In fact, an aspect of AOP saying that a certain procedure or code fragment *a* (for *action* or *advice*) is to be executed with all methods satisfying some predicate *s* (for *selection*) translates to an expression of the form

$$\forall m(x_1,...,x_n) \in M : \\ s(m(x_1,...,x_n)) \rightarrow (m(x_1,...,x_n) \rightarrow a(x_1,...,x_n)) \quad (3)$$

where M corresponds to the set of methods of a program. Note that Eq. 3 is not a first order formula: while a is a first-order predicate specifying the advice of the aspect (the what), s is a second-order predicate selecting certain methods (specifying the where) quantified over the predicate variable $m(\ldots)$. Note that this way the specification of the advice a has access to the parameters of the methods m it applies to (but a need not make use all parameters of m). Without resorting to the second order, the parameters of an aspect cannot be bound to the parameters of the methods they apply to; the aspect remains isolated and hence useless.

Theory aside, it is easy to see that in practice the processing of an aspect requires reasoning about and involves manipulation of a program, that AOP is *de facto* a meta-programming technique (an observation that equally applies to aspect-oriented modelling). On the other hand, in order to actually do something every aspect must contain expressions (method calls etc.) that are on the same level as the items it is an aspect of. Since an aspect always (and necessarily) consists of both, a *what* and a *where* part, there can be no aspect without a meta-level.

On the other hand, postulating that there are (also) aspects in a first-order language (on the same level as other properties, namely types, attributes, relationships, and methods) would either force us to

a) explain what an aspect of an aspect is (or else exclude self-application of the concept), or would
b) require that the *where* part of these aspects applies to propositions one level below the other properties.

As for the latter: both modelling and programming usually start at the level of types; there are no propositions of a lower level so that the subject of first-order aspects would have to remain imaginary. As for the former: the only constellation in which I find aspects of aspects easy to conceive is if aspects are themselves the subject matter. However, these aspects must then be a weaker concept than the aspects of aspect orientation, since there are no aspects they could be applied to (there is no lower level and applying them to themselves or to their second-order relatives would open the door for paradoxes or ill-definedness, as the history of mathematical logic has taught [20]). It follows that first-order aspects are unlikely to exist and, because pure domain models are first order, that these models are aspect free.

4. RELATED WORK

In order to exclude certain paradoxical expressions involving negation and self-reference Russell introduced types to set theory and mathematical logics [20]. His type theory has led to the distinction of first and higher-order logics and—by generalizing the type concept—to the introduction of many and order-sorted logics (the latter being the logical equivalent to the type systems of OOPLs such as C++ and JAVA). Interestingly, as stated before many and order-sortered logics are both first order [9].

Somewhat related to Russell's introduction of types is the work of Tarski and Carnap, who found in their investigations on the concept of truth that when speaking about sentences in a language we must cleanly separate between object and metalanguage [19]. According to this distinction, the former is the language used to speak about objects the in the world, while the later is used for the analysis of the former. Metalanguage is inherently more expressive than object language, since it must contain all sentences of the former plus a notion of truth and corresponding logical operations. Natural language permits paradoxes of Russell's kind only because object and metalanguage are the same. While all languages are products of the mind, the subject matter of object language is the real word, whereas that of metalanguage is itself language and as such un-real (in the literal sense of the word).

Lopes et al. have pointed out that the ability to reference parts of a program (the programmatic equivalence of linguistic anaphora) is a (if not the) key contribution of aspect orientation [8]. Being able to reference what has just been said or done, they argue, is the natural way of keeping specifications both concise and understandable. While I could not agree more with this, I note that this raises the programming language to the level of a metalanguage, since it involves sentences about sentences. The subject matter of these meta-sentences is programming artefacts, which are not themselves objects of the programmed domain.

The relationship of aspects and roles has been investigated by several authors, for instance [4, 5]. Most of this work regards roles as adjunct instances [15], separate objects which are the bearers of role-specific state and behaviour, but whose identity is amalgamated with that of the role player. This would make role-related properties extrinsic to the role-playing object (extrinsic in contrast to its own properties, which are commonly regarded as intrinsic). Contrary

to this view, I argue that the role-playing ability of every object is intrinsic to it, since it must be made possible by its nature. In fact, I prefer to view roles as abstract data types specifying role-related properties and behaviour in the context of one or more collaborations, with the implementation being provided by classes (since different role player classes will implement roles—or provide role-specific features—differently). The role playing of an instance then amounts to that instance being assigned to a variable typed with the role (tantamount to the instance taking part in a collaboration), letting instances pick up and drop roles dynamically. Independent of how roles are being viewed, however, there seems to be consensus that there are only few rather special roles that can be covered by aspects ([4] and Section 2.1).

In contrast to its nature and its role-playing abilities (which, as argued above, should be regarded as the intrinsic properties of an object) aspects in the aspect-oriented sense add extrinsic properties and behaviour, namely features that are attached to objects by reason lying outside their nature.[7] This is why the definition of an aspect can be kept in one place, with second-order expressions specifying where these properties apply. It would appear that properties extrinsic to the objects of a domain are also extrinsic to the domain itself, since the domain consists of only objects and their interactions; one could maintain, though, that it is these interactions aspects focus on, but this has not become evident so far (*cf.* below).

As for the claimed lack of polymorphism of aspects (Section 2.1): Ernst and Lorenz have argued that late binding of advice could be introduced, for instance based on the actual (dynamic) type of the receiver of an intercepted method call [2]. However, Footnote 2 applies in full. In fact, Ernst's and Lorenz's exploration of the possibility to add late bound methods to a statically binding language via aspects ([2, Section 3.5]) is merely a theoretical contemplation and not meant to inspire the design of new programming languages based on late-bound advice rather than methods.

The relationship of aspects and collaborations (of which roles represent the paricipants) mentioned in Section 2.1 also needs further discussion. The definition of an aspect and, in particular, *aspectual collaborations* [7] can involve roles, but these roles are not themselves aspects. Surely, one could argue that if roles are valid modelling elements, then it is hard to see why an aspect defining the roles should not equally be considered as a domain-level concept. In fact, a collaboration of objects is identifiable at the same level as the objects themselves, and generalizing it (by introducing role types as placeholders for role players) does not raise it to a meta-level: for instance, `Printing` is a collaboration that is on the same (domain) level as its roles `Printer` and `Printed`. However, even though blending of collaborations and aspects is possible [7], the two are not the same concept (after all, not all aspects involve roles); a `Printing` aspect for instance would be largely infeasible, since the knowledge of how to print/be printed is intrinsic to the role-playing objects. The aspect could serve as a reification of the collaboration, but this does not seem to be what aspects were intended for. All that remains is to add extrinsic behaviour, which is likely to be extrinsic to the problem as well.

On a wider scope other authors have suggested that aspects are not only useful for programming, but also for the earlier software development phases including analysis and requirements engineering (e.g., [1, 12]). However, despite several announcements to the opposite all examples presented so far seem to be concerned with non-functional (rather than functional) requirements and as such pertain to the solution of a problem, not to the problem domain. This apparent shortage of examples of *functional aspects* could be explained by the fact that most domain models are indeed aspect free.

Of course my position could be proven wrong simply by providing the counterexamples that are announced here and there. However, I would conjecture that finding such examples is not as straightforward as it might seem, since in order to be sufficient a counterexample must fulfil the following criteria:

- the aspect must be an aspect in the aspect-oriented sense (in particular, it must not be a role);
- it must not be an artefact of the (technical) solution, but must be seen as representative of an element in the underlying problem domain; and
- its choice must have a certain arbitrariness about it so that the example provides evidence that there are more aspects of the same kind, be it in the same or in other domains.

5. CONCLUSION

Aspect-orientation has set off to augment all phases of software engineering—and their artefacts—with the notion of an aspect. This would include the analysis phase and with it object-oriented modelling of a problem domain. Although an actual proof would require more rigorous reasoning (including a complete and agreed upon formalization of both domain models and aspects), I believe to have made plausible that domain models are, under reasonable preconditions, aspect free. This is in contrast to some of the published literature, which seems to suggest that so-called functional aspects exist in the same right and frequency as their more popular, non-functional siblings. So far, I have not come across any convincing examples of aspects of this kind; however, I will gladly accept and discuss any suggestion thereof.

[7] Note that aspects can be used to implement adapters for classes (or entity types, see e.g. [11]) but this can also be done with adapter classes and makes sense only if the aspect weaver is more flexible than the compiler.

6. REFERENCES

[1] J Araújo, A Moreira, I Brito, A Rashid "Aspect-oriented requirements with UML" *Second International Workshop on Aspect-Oriented Modelling with UML* (2002).

[2] E Ernst, DH Lorenz "Aspects and polymorphism in AspectJ" in: *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development* (ACM 2003) 150–157.

[3] M Fowler *Refactorings: Improving the Design of Existing Code* (Addison-Wesley, 1999).

[4] KB Graversen, K Østerbye "Aspect modelling as role modelling" in: *OOPSLA '02 Workshop on Tool Support for Aspect Oriented Software Development* (2002).

[5] EA Kendall "Role model designs and implementations with Aspect-Oriented Programming" in: *OOPSLA* (1999) 353–369.

[6] BB Kristensen, K Østerbye "Roles: conceptual abstraction theory and practical language issues" *TAPOS* 2:3 (1996) 143–160.

[7] KJ Lieberherr, DH Lorenz, J Ovlinger "Aspectual collaborations: combining modules and aspects" *The Computer Journal* 46:5 (2003) 542–565.

[8] CV Lopes, P Dourish, DH Lorenz, K Lieberherr "Beyond AOP: toward naturalistic programming" in: *OOPSLA'03 Special Track on Onward! Seeking New Paradigms & New Thinking* (ACM 2003) 198–207.

[9] A Oberschelp "Untersuchungen zur mehrsortigen Quantorenlogik" *Mathematische Annalen* 145 (1962) 297–333.

[10] OMG *http://www.uml.org/*

[11] A Rashid, P Sawyer, "Aspect-orientation and database systems: an effective customisation approach" *IEE Proceedings – Software* 148:5 (2001) 156–164.

[12] A Rashid, P Sawyer, AMD Moreira, J Araújo "Early aspects: a model for Aspect-Oriented Requirements Engineering" *RE* (2002) 199–202.

[13] T Reenskaug, P Wold, OA Lehene *Working with Objects – The OOram Software Engineering Method* (Addison-Wesley 1996).

[14] J Richardson, P Schwartz "Aspects: extending objects to support multiple, independent roles" in: J Clifford, R King (eds) *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data* SIGMOD Record ACM Press, 20:2 (1991) 298–307.

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

[16] F Steimann "A radical revision of UML's role concept" in: A Evans, S Kent, and B Selic (eds) *UML 2000, Proceedings of the 3rd International Conference* (Springer 2000) 194–209.

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

[18] F Steimann, T Kühne "A radical reduction of UML's core semantics" in: JM Jézéquel, H Hussmann, S Cook *UML 2002: Proceedings of the 5th International Conference* (Springer, 2002) 34–48.

[19] A Tarski "The semantic conception of truth and the foundations of semantics" *Philosophy and Phenomenological Research* 4 (1944).

[20] AN Whitehead, B Russell *Principia Mathematica* (Cambridge University Press, 1910).