Domain Models Are Aspect Free

Friedrich Steimann

Fachbereich Informatik, Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
Universitätsstraße 1, D-58097 Hagen
steimann@acm.org

Abstract. Proponents of aspect orientation have successfully seeded the impression that aspects—like objects—are so fundamental a notion that they should pervade all phases and artefacts of the software development process. Aspect orientation has therefore proliferated from programming to design to analysis to requirements, sparing neither software processes nor their favourite languages. Since modelling plays an important role in software engineering, much effort is currently being invested in making modelling languages aspect ready. However, based on an observed lack of examples for domain level (or functional) aspects this paper argues the case against the omnipresence of aspects, particularly the existence of aspects in domain models, and offers some informal arguments as well as a semiformal proof in favour of the claims made.

1 Introduction

Since the term AOP came public at ECOOP in 1997 [1], 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 have 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 and PhD thesis 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

[1] Popular precursors of (and contributors to) the AOP paradigm were Composition Filters [1], DEMETER [15], and Subject-Oriented Programming [12].

L. Briand and C. Williams (Eds.): MoDELS 2005, LNCS 3713, pp. 171-185, 2005.

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 single, rather special problem domain.

The remainder of this paper is organized as follows. First I will 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 domain model, i.e., do not refer directly to the modelled domain. Based on these findings I will 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 claim 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 found in software modelling. The discussion may be incomplete, yet I believe it covers the most important points being taken in the literature, and suffices to show that these kinds of aspects are either not needed for domain modelling, or lie outside its scope.

2.1 Aspects as Roles

Long before the term aspect-oriented programming was coined, it was discovered that objects can have different *facets*, *views*, *perspectives*, *roles*, or *aspects* [24]. The classic example of a class whose instances have many roles[2] is *Person*: *Employee*, *Employer*, *Customer*, *Student*, and so forth are all roles that can be played by a person. 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 ad-

[2] 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), I use the term *role* here.

junct instances [24]. All share the same least intent: to let objects of *same type* have *different properties* in different contexts at different times.

There is however another important characteristic of the role concept: 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 unrelated 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 – the roles are in fact polymorphic, meaning that they have different implementations. Factoring out different implementations to a single place as suggested by an aspect-oriented approach would seem inapt, since it would contradict basic object-oriented principles.[3] Instead, interfaces (specifying protocol, but lacking implementation) and multiple (interface) inheritance readily lend themselves to representing roles and role playing, respectively, with mixins stepping in to allow for the inheritance of code wherever deemed appropriate [25, 26].

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 [2, 17, 25]. 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 independently of one another; in fact, the obliviousness property of aspect orientation [8] suggests that aspects have no mutual knowledge of each other.[4] It follows immediately that modelling the roles of a system as aspects works only in cases where roles are isolated and monomorphic.[5]

---

[3] In fact, it would effect to replacing the polymorphism of a role with a conditional (reversing the *Replace Conditional with Polymorphism* refactoring [9]): code treating objects of different types differently would not be attached to the types, but located in a single place, a conditional (typically a switch statement) branching on the type of an object. Although aspects could be made polymorphic [4], doing so does not better the situation, since the definition of role-playing objects would remain scattered.

[4] Note that aspects can apply to aspects, so that there may be some unilateral awareness. Also, aspects can model collaborations [2, 25], which includes the modelling of roles; the roles themselves however are no aspects.

[5] 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, particu-

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 [23]) 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 could help without major modifications. Aspectual collaborations [15] address these problems in some detail, but use roles in the specification of aspects, without equating the two concepts (*cf*. related work in Section 5).

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. An aspect on the other hand is neither a type, nor is it meaningful only in the context of another aspect, nor does is naturally introduce different implementations for different objects. 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.[6]

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 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 [19] 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 and thus avoiding the dominance of one structure (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

larly 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 in Section 5.2 for more on this issue.

[6] A contrary, but not very convincing view is held in [11].

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, which means 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.[7]

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.[8] 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 the term *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).

---

[7] 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 these ordering dimensions are themselves part of the ordered domain.

[8] Having said this, we note that sometimes a technical aspect has a namesake in the problem domain: in the perennial ATM example, for instance, transactions and logs are entities that occur in the problem domain. However, these entities 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 considered (and implemented) as ordinary types.

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 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 not of the modelled problem). 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, this provides no evidence that there are aspects *in* the domain being programmed or modelled, unless in the rather special case that the modelled domain is *Modelling* itself.

As an aside, the fact that modelling has aspects implies that it requires some kind of weaving. In fact, since every model (model here defined as a single diagram) usually specifies only one tiny aspect of a modelled problem. 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 a problem domain. This issue is picked up again in Section 4.

2.5 Aspects as Non-functional Requirements

Those who have given up on searching for functional aspects (or perhaps never did so) have retreated to the position that aspects model non-functional requirements. Non-functional requirements are often considered to be hard to express given the usual modelling languages (which might explain their absence from domain models); however, this is not necessarily so. For instance, that a banking transaction may only take a certain period of time would require that the modelling language has a notion of time, which is nothing too special in disciplines other than software engineering. Likewise, that a money withdrawal requires authentication can be expressed through an ordinary sequence diagram. In fact, that something is classified as a non-functional requirement does not preclude it from being part of a domain model – rather, it is the fact that it cannot be reified.

Modelling languages are usually first-order languages [5, 20]. This implies that statements about statements cannot be expressed unless the statements themselves become objects, that is, are reified. Aspects on the other hand are typically expressed as statements quantified over an infinite number of statements; in fact, non-functional

requirement that might be expressed by an aspect are usually of the form "for all functional requirements of kind *x*, make sure that *y*". For instance, a statement of the form "make sure that all methods of a program that are called in the course of a transaction are logged" is something that cannot be expressed using the means of a first-order language, as will be argued below.

To conclude, one could be led to argue that aspects invariably express non-functional requirements, so if non-functional requirements are no elements of domain models, then neither are aspects. But even if one dismisses this argumentation (because certain non-functional requirements *can* be expressed using standard modelling languages), this does not imply that aspects can be found in domain models, since not all non-functional requirements are adequately expressed as aspects. In fact, as will be argued next, it is the very nature of aspects that makes them unsuitable for being included in domain models.

3 Proving Aspect-Freeness of Domain Models

Given that roles have properties that make them unsuitable for being modelled as aspects, that ordering dimensions are 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 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 is really needed is a positive argument making the claimed non-existence plausible or, better still, a proof of thereof.

Obviously, such a proof depends critically on two definitions: what a domain model is, and what an aspect is. Since both terms are in a rather broad use, definitions that are both precise and generally accepted are hard to find. I will therefore attempt a semiformal proof that builds on preconditions that should be easy to accept for a wide audience. That such a proof must remain debatable is a tribute to the diversity of the work in the field, and the many views held by the many authors. However, the proof should be seen in light of the observed absence of domain-level, or functional, aspects and as such as an explanation attempt in the tradition of natural science; questioning its soundness only leaves the observations unexplained, it does not make them wrong.

3.1 The First-Orderedness of Domain Models

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 free of propositions

(it is only entities that exist), unless of course "reality" (the modelled domain) is language.

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 domain model contains no propositions about propositions, since these would describe the model rather than reality. Generally, there is broad consensus that *domain models are first order* (e.g., [5, 20]). Indeed, it appears that 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. [27]

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

$$\forall o : T(o) \rightarrow U(o) \tag{1}$$

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 (1) occur repeatedly in object-oriented models (they express the type hierarchy), it is commonplace to introduce a special relationship, called generalization, whose instances (tuples) relate types (and thus predicates) rather than objects. In fact, in a model we would not write (1), but

$$T < U \tag{2}$$

or something alike. However, generalization as a relationship is only *extensionally* defined (i.e., by listing all its elements) – it rolls out to a finite set of first-order formulas in the style of (1).[9] And indeed, even though (2) suggests that that type T inherits the properties from type U, it is only the declaration of properties that is inherited (where the properties themselves pertain to the types' objects).

It is an interesting result of mathematical logic 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) [18]. Thus, the fact that a model is object-oriented does not negate that it is a pure domain model in the above sense. As it turns out, this is generally not the case for as-

[9] In particular, generalization does not quantify over types (*cf.* Footnote 11 for a contrary position).

pect-oriented models, which typically quantify over open (potentially infinite, in any case *intensionally* defined) sets of propositions (*cf.* related work in Section 5, in particular [8]).

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* (or *when*) 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

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

where *M* corresponds to the set of methods of a program. Note that (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*(…). 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; this applies equally 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*/*when* 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 ori-

entation, 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 [29]). 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 Possible Impact of Aspect Orientation on Domain Modelling

The immediate (and also rather dramatic) consequence of the absence of aspects from first-order languages is that it frees all modelling languages that are (and are to remain) first order from having to introduce aspects as an additional modelling construct. This may come as a disappointment to some, but should really be perceived as a relief rather than a setback, as the following argumentation shows.

The main advantage of graphical models (diagrams) over programs (text) is that they can express proximity in more than one dimension. In fact, literally all diagrams use lines to indicate the relatedness of concepts (represented by boxes and other shapes), thereby distinguishing conceptual proximity from the geometric one that results from diagram layout.[10] However, aspect orientation breaks with the proximity ("locality") concept of a language [8], so that the principal advantage of graphical over textual notations is lost. This explains why there seems to be no natural way of integrating aspects into UML as a complementary concept (see, e.g., [3, 4, 13] for attempts), an observation that should really come as no surprise, for subroutines (another language construct that breaks with locality [8]) cannot be represented naturally in flowcharts either. Seen this way, that everything can remain as is—at least for domain models—is good news.

Things get different, however, as soon as we switch from domain modelling to metamodelling. Metamodelling requires a second-order language (a language that can make statements about a language; *cf.* Section 5), in which aspects can be expressed. This might turn out to be extremely handy.

As mentioned in Section 2.4, modelling itself has many aspects; it could in fact be considered aspect oriented. An aspect language could be devised that allows one to model modelling much more adequately than the metamodelling languages used today (e.g., MOF or even UML); that allows the integration of functional and non-functional views, of static and dynamic views, of analysis, design, and even deployment views (which all could be considered aspects in this aspect-oriented metamodelling language) by suitable weaving techniques. The definition of such a metamodelling language would include aspects as a modelling concept but, as argued above, each concrete aspect would be a construct of the modelling language, not any domain modelled with it. It follows that only if a modelling language is itself considered the domain of modelling is it possible that we have an aspect *in* the domain. However, the discussion of metamodels and their languages is not what this paper is about.

[10] That related elements of a diagram are mostly also in geometric proximity of each other is a tribute to readability, but neither necessary nor always possible.

5 Related Work

5.1 Aspects and Second Order

In order to exclude certain paradoxical expressions involving negation and self-reference Russell introduced types to set theory and mathematical logics [29]. 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 pendant to the type systems of OOPLs such as C++ and JAVA). Interestingly, as stated before both many and order-sorted logics are first order [18].

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 [28]. 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). Thus, metalanguages are not needed to model reality and, more important for the claim of this paper, concepts that can only be expressed by means of a metalanguage are not found in the modelled domain.

Filman and Friedman have identified "quantified programmatic assertions" ("quantification") as a "distinguishing characteristic of AOP" [8]. As it turns out, (3) is a formal paraphrase of their sentence

> "In programs *P*, whenever condition *C* arises, perform action *A*" [8] (4)

where *P* corresponds to *M* in (3), *C* corresponds to *s*(.), and *A* to *a*(…). That *C* is formulated in terms of (the elements of) *P* and thus second order is implicit in the surrounding text; obliviousness, the other defining characteristic of AOP, is also an implicit consequence of (4), since the elements of *P* have no knowledge of the conditions *C*. According to Filman and Friedman, no language (construct) that lacks quantification or obliviousness can be called aspect-oriented; since quantification involves second-order statements, first-order languages are aspect free.[11]

Lopes et al. have also 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 [17]. 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 program-

[11] Deviating from my argumentation in Section 3.1, the authors view mixins and even general inheritance as a form of quantification, since it induces statements of the form "for all classes inheriting from me, add …". However, neither programs nor models actually quantify over the inheritance relationship; instead, they include explicit statements of inheritance so that the "quantification" is in fact a finite (and explicit) conjunction; in particular, as argued in Section 3.1, it is not second order.

ming 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.

5.2 Aspects and Roles

The relationship of aspects and roles has been investigated by several authors, for instance [10, 11, 14]. Most of this work regards roles as adjunct instances [24], 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 would be 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 ([10] 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.[12] 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 [4]. However, Footnote 3 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 ([4, 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 deserves further discussion. The definition of an aspect and, in particular, *aspectual collaborations* [15] 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

[12] Note that aspects can be used to implement adapters for classes (or entity types, see e.g. [20]) but this can also be done with adapter classes and makes sense only if the aspect weaver is more flexible than the compiler.

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 [15], 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.

5.3 Early Aspects

Some authors (e.g., [1, 4, 22]) suggest methods for the discovery and handling of aspects in the non-functional and functional requirements of a software product ("early aspects"). However, the language of requirements is largely informal, as is the authors' notion of an aspect. That a functional requirement crosscuts several others does not suffice for it to be considered an aspect, at least not in the strict sense (such as elaborated here or in [8], as reflected in (3) and (4)). Instead, one could argue that "obliviousness" [8] is hardly a required property of a functional requirement, and that all "quantification" in the requirements list is over this (finite) list of requirements so that neither of the defining criteria of [8] for aspects is fulfilled. In fact, "candidate aspects" identified at the functional requirements level are formally indistinguishable from roles or plain old subroutine calls, and the claim of this paper is that in a domain model, they end as such.

6 Falsification of My Thesis

Of course my position could be proven wrong simply by providing counterexamples. 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 subroutine or 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.

7 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 a full proof would require more rigorous reasoning (including complete formal definitions of both domain models and aspects, and widespread acceptance of these definitions), I believe to have made plausible that domain models are, under generally accepted 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. As a result of my argumentation, domain modelling is freed from the felt obligation to become aspect oriented.

Acknowledgments

The paper has profited from helpful comments from various anonymous reviewers. Thank you for taking the time!

References

1. M Aksit, L Bergmans, S Vural "An object-oriented language-database integration model: the composition-filters approach" in: *ECOOP '92* (1992) 372–395.
2. M Aksit, K Wakita, J Bosch, L Bergmans, A Yonezawa "Abstracting object-interactions using composition-filters" in: R Guerraoui, O Nierstrasz, M Riveill (eds) *Object-Based Distributed Processing* ECOOP '93 Workshop, Springer LNCS 791 (1994) 152–184.
3. J Araújo, A Moreira, I Brito, A Rashid "Aspect-oriented requirements with UML" *Second International Workshop on Aspect-Oriented Modelling with UML* (2002).
4. ELA Baniassad, S Clarke "Theme: an approach for aspect-oriented analysis and design"in: ICSE 2004 (2004) 158–167.
5. J Edwards, D Jackson, E Torlak "A type system for object models" in: RN Taylor, MB Dwyer (eds.) *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (ACM 2004) 189–199.
6. T Elrad, O Aldawud, A Bader "A UML profile for aspect oriented modeling" in: *OOPSLA 2001 workshop on Aspect Oriented Programming* (2001).
7. 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.
8. RE Filman, DP Friedman "Aspect-oriented programming is quantification and obliviousness" in: *OOPSLA Workshop on Advanced Separation of Concerns* (Minneapolis, 2000).
9. M Fowler *Refactorings: Improving the Design of Existing Code* (Addison-Wesley, 1999).
10. KB Graversen, K Østerbye "Aspect modelling as role modelling" in: *OOPSLA '02 Workshop on Tool Support for Aspect Oriented Software Development* (2002).
11. S Hanenberg, R Unland "Roles and aspects: similarities, differences, and synergetic potential" in: Z Bellahsène, D Patel, C Rolland (eds) *OOIS 2002* Springer LNCS 2425 (2002) 507–520.
12. WH Harrison, H Ossher "Subject-oriented programming (a critique of pure objects)" in: *8th OOPSLA* (1993) 411–428.

13. M Kande, J Kienzle, A Strohmeyer *From AOP to UML: towards an aspect-oriented architectural modeling approach* Technical Report, Swiss Federal Institute of Technololgy (Lausanne, 2003).
14. EA Kendall "Role model designs and implementations with Aspect-Oriented Programming" in: *OOPSLA* (1999) 353–369.
15. KJ Lieberherr, AJ Riel "Demeter: a case study of software growth through parameterized classes" in: *10th ICSE* (1988) 254–264.
16. KJ Lieberherr, DH Lorenz, J Ovlinger "Aspectual collaborations: combining modules and aspects" *The Computer Journal* 46:5 (2003) 542–565.
17. 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.
18. A Oberschelp "Untersuchungen zur mehrsortigen Quantorenlogik" *Mathematische Annalen* 145 (1962) 297–333.
19. OMG *http://www.uml.org/*
20. B Paech, B Rumpe "A new concept of refinement used for behaviour modelling with automata" in: M Naftalin, BT Denvir, M Bertran (eds.) 2nd International Symposium of Formal Methods Europe Springer LNCS 873 (1994) 154–174.
21. A Rashid, P Sawyer, "Aspect-orientation and database systems: an effective customisation approach" *IEE Proceedings – Software* 148:5 (2001) 156–164.
22. A Rashid, P Sawyer, AMD Moreira, J Araújo "Early aspects: a model for Aspect-Oriented Requirements Engineering" *RE* (2002) 199–202.
23. T Reenskaug, P Wold, OA Lehene *Working with Objects – The OOram Software Engineering Method* (Addison-Wesley 1996).
24. F Steimann "On the representation of roles in object-oriented and conceptual modelling" *Data & Knowledge Engineering* 35:1 (2000) 83–106.
25. 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.
26. F Steimann "Role = Interface: a merger of concepts" *Journal of Object-Oriented Programming* 14:4 (2001), 23–32.
27. 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.
28. A Tarski "The semantic conception of truth and the foundations of semantics" *Philosophy and Phenomenological Research* 4 (1944).
29. AN Whitehead, B Russell *Principia Mathematica* (Cambridge University Press, 1910).