Domain Models are NOT Aspect Free

Awais Rashid, Ana Moreira

Computing Department, Lancaster University, Lancaster LA1 4WA, UK
awais@comp.lancs.ac.uk

Departamento de Informática, Universidade Nova de Lisboa, 2829-516 Lisboa, Portugal
amm@di.fct.unl.pt

Abstract. In proceedings of MoDELS/UML 2005, Steimann argues that domain models are aspect free. Steimann's hypothesis is that the notion of aspect in aspect-oriented software development (AOSD) is a meta-level concept. He concludes that aspects are technical concepts, i.e., a property of programming and not a means to reason about domain concepts in a modular fashion. In this paper we argue otherwise. We highlight that, by ignoring the body of work on Early Aspects, Steimann in fact ignores the problem domain itself. Early Aspects techniques support improved modular and compositional reasoning about the problem domain. Using concrete examples we argue that domain models do indeed have aspects which need first-class support for such reasoning. Steimann's argument is based on treating quantification and obliviousness as fundamental properties of AOSD. Using concrete application studies we challenge this basis and argue that abstraction, modularity and composability are much more fundamental.

1. Introduction

As new software development paradigms appear on the horizon, it is normal that debates rage over their merits and demerits. Aspect-oriented software development (AOSD) [13] is no stranger to this situation. Since Kiczales et al's invited paper at ECOOP'97 [22], several points and counterpoints have been made in literature arguing about the merits and demerits of modularising crosscutting concerns in separate abstractions. Over the years the focus of aspect-orientation has significantly expanded beyond programming. A number of aspect-oriented analysis and design approaches, e.g., [2, 6, 17, 26, 27, 34, 39], aimed at disentangling requirements, architecture and design descriptions have appeared. These approaches provide explicit support for identification, modular representation, composition and analysis of broadly-scoped properties of both a functional and non-functional nature. In fact, several approaches, e.g., [8, 27, 38], take a multi-dimensional perspective on the problem and remove the strong distinction between aspects and the concerns they crosscut. Thus they also remove any distinction about whether a concern is functional or non-functional hence facilitating uniform modelling of concerns and their crosscutting influences (amidst other dependencies and interactions).

In his MoDELS/UML 2005 paper [35], Friedrich Steimann, however, argues that AOSD approaches in general, and aspect-oriented analysis and design approaches in particular, are merely useful for representing meta-level concepts. His hypothesis is that aspects are second order entities that only require meta-modelling support and that domain models are in fact aspect free. His overall conclusion is that aspects are technical concepts, i.e., a property of *programming*, and not a means to reason about domain concepts in a modular fashion. In other words: there are no *functional* aspects and non-functional aspects are properties of the solution domain that do not require first-order representation. In his discussion, Steimann disregards the work on Early Aspects indicating that just because a functional requirement crosscuts other requirements does not mean that it should be treated as an aspect. Steimann's notion of an aspect is rooted in the properties of *quantification* and *obliviousness* as proposed by Filman and Friedman [14]. He treats these as fundamental properties of any aspect-oriented approach and, on this basis, argues about the second-ordered nature of aspects – to paraphrase Steimann: aspects are meta-level concepts that manipulate base-level (or first-order) elements.

In this paper we argue otherwise. We contend that if one is to discuss whether a domain model has crosscutting concerns, one cannot disregard the problem descriptions themselves. Therefore, we base our argument on Early Aspects techniques which support improved modular and compositional reasoning about the problem domain. Using concrete examples rooted in these techniques we argue that domain models do indeed have aspects which need to be modularised effectively to enable us to reason about them in a modular fashion. Similarly, using concrete application studies we challenge the fundamental basis of Steimann's argument, i.e., the notion of quantification and obliviousness. We demonstrate that *abstraction*, *modularity* and *composability* are much more fundamental to AOSD than quantification and obliviousness (which, though desirable are not necessary defining characteristics of an *aspect*). We conclude by discussing that, even if quantification and obliviousness were to be considered fundamental, firstly, early aspects techniques meet these characteristics and, secondly, the notion of aspects in the problem domain, as demonstrated by Early Aspects techniques, flows into the solution space, requiring first-class modelling of functional and non-functional aspects.

The rest of the paper is structured as follows. Section 2 lists Steimann's main four perspectives that give body to his argument. Section 3 debates each of these four arguments, showing counter examples. Section 4 explains why quantification and obliviousness cannot be understood as necessary defining properties of an aspect, discussing other equally valid views not aligned with Filman and Friedman's perspective. We argue that, just like with other separation of concerns approaches, abstraction, modularity and composability are the fundamental characteristics of AOSD. In Section 5 we discuss how first-class aspects, both functional and non-functional, in the problem domain flow into the solution space hence requiring their first class representation in the solution domain. Finally, Section 6 concludes the paper by discussing how our argument invalidates Steimann's hypothesis while still satisfying several constraints set by him.

2. Steimann's Argument

Steimann's argument about domain models being aspect free is based on four different perspectives:

1. Relationship between the notion of an aspect and a role;
2. The lack of any observed examples of arbitrary functional aspects in the current literature;
3. Aspects being strictly non-functional properties that are in fact aspects of the solution rather than the problem domain;
4. The second-order nature of aspects, i.e., aspects must always manipulate entities in a first-order separation.

From the above four perspectives, Steimann argues that for functional aspects to exist, and hence the need for them to be modelled, they must be at the same level of abstraction as other elements in the domain. Using a semi-formal proof based on *quantification* and *obliviousness* [14] he argues that aspects are always second-order statements that manipulate first-order elements thus concluding that they are meta-level concepts. From this semi-formal proof he also draws his conclusion that no functional (or domain) aspects exist.

We discuss quantification and obliviousness in detail in Section 4. Before that, in section 3, we debate each of the above four perspectives underpinning Steimann's argument. As mentioned above, Steimann disregards the work on Early Aspects stating that natural language descriptions are too imprecise to be aspectised. However, stakeholders, who are the primary descriptors of a problem domain, tend to specify their problems using natural language. These natural language descriptions are where aspects first manifest themselves as broadly-scoped properties leading to tangled representations in requirements models and subsequently in architecture, design and implementation. If we are to look for the existence of functional aspects in domain models we must start at the requirements analysis stage. Thus, this is where we start our search for aspects in domain models.

3. Aspects in Domain Models

When discussing the existence of aspects in domain models, we first examine Steimann's perspective on aspects and roles. In subsection 3.1, we demonstrate that his perspective is just one observation on the relationship between the two concepts and other equally valid arguments exist that demonstrate the synergy between the two concepts and their mutual complementarity. Then, in subsection 3.2, we show evidence, by means of practical examples drawn from the body of work on Early Aspects, that functional aspects do exist and can be found in everyday problems. In subsection 3.3, we discuss that non-functional requirements are not just properties of the solution but in fact properties of the problem that, too, require first-class modelling support. Finally, in subsection 3.4, we provide additional arguments as to why aspects require a first-order representation.

3.1 On the Relation between Aspects and Roles

Steimann *equates* an aspect to a role. He argues that for roles to be appropriately realised, each object must explicitly implement all the roles it intends to play. In his view, since most role implementations tend to be specific to the particular class of objects, it is not reasonable to assume that role implementations can indeed be aspectised. This is, however, not the case. Several roles can be very generic. Most design patterns utilise the notion of roles to decouple the pattern implementation from its concrete usage in a specific application. For instance, the Observer pattern uses the Subject and Observer roles for this purpose. A number of design modelling approaches, e.g., Theme/UML [8] have shown how aspect-oriented techniques can be employed to improve the modular representation of design patterns such as the Observer pattern. Similarly, Hannemann and Kiczales [16] have demonstrated how design pattern implementations can benefit from the use of aspect-oriented programming (AOP) in terms of code locality, reusability, composability and pluggability. Garcia et al. have used these implementations as a basis of their quantitative evaluation of the benefits and scalability of AOP [5, 15]. Their studies show significant improvements in the case of 13 out of 23 design patterns with regards to metrics such as separation of concerns, coupling, cohesion and size. These studies mostly represent roles as interfaces with the glue code, between their abstract representation in the modularised pattern implementation and its concrete application instantiation, being provided through aspect-oriented composition mechanisms. This relationship between roles and aspects is entirely different from what is perceived by Steimann. Roles remain completely polymorphic as they are realised through interfaces while aspects provide the modularity and composition support essential to modularise the pattern implementation in a separate *aspectual component*.

Kendall's work [20] demonstrates a similar yet orthogonal relationship. She utilises AOP as a means to improve the implementation of role models. Through re-engineering of an existing role-based framework to an AspectJ implementation, she demonstrates that an aspect-oriented implementation is more cohesive than an object-oriented one.

Hannemann and Kiczales as well as Kendall utilize AOP as a means to improve the modularity of role-based implementations. Another different, yet equally valid, perspective arises from the ability of roles to help us realise multi-faceted objects. Roles can apply (often dynamically) across the system and hence, role-based systems tend to be less prescriptive about how objects interact. This ability makes it possible for role-models to facilitate aspect composition as is the case in CaesarJ [28]. In this case the *provided* and *required* interfaces specify the roles an aspect can play in a composition and those it expects of other modules in the system.

Steimann further argues that roles are polymorphic by nature and aspects are not. This is not true. Firstly, most aspect-oriented approaches facilitate aspect inheritance hence respecting the substitutability semantics that are normal in object-oriented hierarchies. Though approaches such as AspectJ [1] restrict the programmer to implicit aspect instantiation through the language framework, other techniques, e.g., CaesarJ [28], Composition Filters [4], JBoss [18] and Vejal [33], facilitate explicit aspect instantiation hence supporting substitutability of an aspect instance of a sub-aspect-type whenever an instance of a super-aspect-type is required. Since most of

these approaches reify aspects as first-class objects (or use Java classes to specify aspect behaviour with XML descriptors specifying the aspect compositions), any role realisation using such AOP mechanisms can have the same polymorphic nature as a pure OO role realisation. It is perfectly conceivable that using an approach such as Composition Filters one can have a core object with a set of attached filters, each of which realises a specific role the object has to play (cf. Figure 1 – note only incoming message filters are shown but similar logic applies to outgoing messages). The per instance attachment ability of Composition Filters further facilitates an object-specific (unlike class-specific implementation in most standard OO techniques) configuration of roles that an object may participate in – this has been realised in the context of implementing roles at each edge of association and aggregation relationships using the SADES implementation of composition filter concepts [31]. Since such filters are implemented as first-class elements, polymorphic properties of roles are fully preserved.

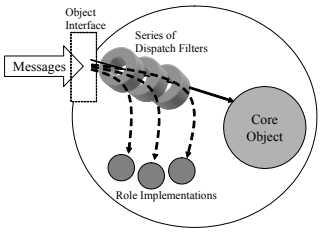

Figure 1: Polymorphic Role Implementations with AOP using Composition Filters

Having established the complementary nature of roles and aspects, we can also say that aspects do exist in domain models. Roles are a domain concept. Different objects in different domains play different (perhaps sometimes overlapping) sets of roles. Since roles have a broadly-scoped nature and aspects can be used to realise role models in a fashion that supports role modularity without compromising role polymorphism, aspects do exist in domain models. However, one might argue that roles naturally form good candidates for aspects. In a system not following a role model design principle, are there indeed crosscutting functional and non-functional properties that are first-order domain elements? We discuss this next.

3.2 Observed Examples of Arbitrary Functional Aspects

For such observed examples, we turn to the extensive body of work on Early Aspects [2, 6, 8, 17, 26, 27, 34, 39]. Steimann disregards the work in this space by stating that the language of requirements is informal and that aspect-oriented requirements engineering approaches do not satisfy the quantification and obliviousness properties. Requirements engineering is mainly concerned with reasoning about the problem domain and formulating an effective understanding of the stakeholders' needs. Such an understanding leads to the emergence of a requirements specification that forms a bridge between the problem domain and the solution domain, the latter being the system architecture, design and implementation. So if one is to argue about the existence of aspects in domain models, one must examine the body of work in Early

Aspects and specifically that on aspect-oriented requirements engineering. Though we argue in Section 4 that quantification and obliviousness are desirable, not fundamental, properties of AOSD approaches, Steimann's assertion that Early Aspects techniques do not satisfy these properties is incorrect. In fact, several approaches, e.g., [2, 6, 27, 34, 39], do not require any specific hooks within the base decomposition hence satisfying the obliviousness property. Furthermore, they have powerful composition mechanisms based on high-level declarative queries and semantics-based join point models that certainly do satisfy the quantification property. Figure 2 shows simplified viewpoint and aspect definitions as well as an example composition specification in the viewpoint-based aspect-oriented requirements engineering approach we presented in [34] – note we omit the XML notation for simplification. The problem domain in question is that of online auction systems. We can observe that the base concerns, i.e., the viewpoints Seller and Buyer are oblivious of the aspect Bidding whose associated composition specification quantifies over a set of viewpoint requirements to which it applies. Incidentally, note that the aspect Bidding is a core functional property of the system and not a non-functional one.

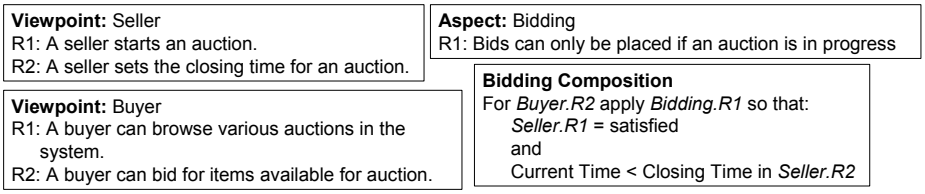

Figure 2: Obliviousness in viewpoint-based aspect-oriented requirements engineering

Reasoning about the problem domain with early aspects

Let us look at the specific problem description of an online auction system and analyse what are the various crosscutting functional and non-functional concerns. We use a viewpoint-based requirements specification mechanism. Aspects in the specification crosscut the viewpoints, each of which represents requirements from a specific stakeholders' perspective. The viewpoints in this specific problem description are also analogous to roles as they capture the requirements about specific user roles, i.e. the System Administrator, Customer, Seller, Buyer, System Owner and Webmaster. As shown in Figure 3, such a system has a number of concerns that crosscut the requirements of these various viewpoints (or roles). For instance, the *bidding* aspect affects the customer viewpoint because customers are interested in bidding for the items being auctioned. It also affects sellers as they are the primary stakeholders interested in the bids. At the same time, the system administrator is interested in ensuring that bids are only received until the specified auction closing time, and so on. The same is true of the *selling* aspect which affects these multiple viewpoints. Another aspect of significance is the *bid solvency* concern which dictates that all placed bids must be solvent, i.e. a customer must have more credit than the sum total of all the bids s/he has in progress. This is of key concern to the system administrator and owner as they wish to ensure that sellers recover their due payments. At the same time, this is a key factor in the seller choosing the specific

auction system for the security and trust the bid solvency aspect offers. All the aspects, i.e. bidding, selling, bid solvency, etc. are functional properties of the domain hence requiring first-class modelling support. They are not *properties of the program* to be developed to satisfy the auction system requirements. Nor are they second order entities as the various viewpoints have strong dependencies on the semantics of these aspects and are at the same level of abstraction as the aspects themselves.

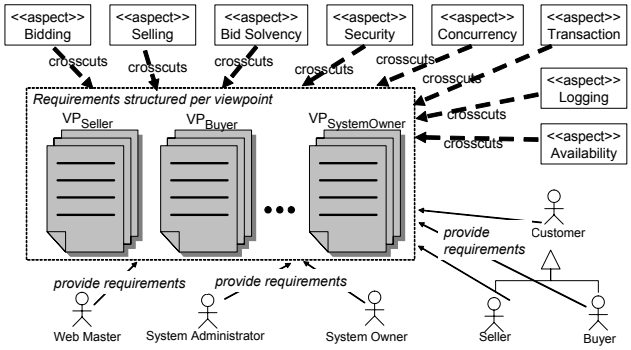

Figure 3: Aspects in a viewpoint-oriented model of the auction system

Other evidence in existing literature

Jacobson and Ng [17] offer an aspect-oriented use case approach to handle stakeholder concerns from requirements analysis through to low-level design. Their proposal is based on the observation that use cases reflect stakeholders' concerns and are crosscutting by nature. Therefore, each use case is encapsulated in a use-case module which typically contains one non-use-case specific slice (that only adds classes to the module) and one or more use-case slices which contain classes and aspects specific to the realisation of the use case. It is worth observing that use-case slices (and, therefore, aspects) identified in this work are typical functional aspects and may represent extensions, inclusions and certain secondary flows used in classical object-oriented modelling, which makes use case slices abundant for each new problem. In their hotel reservation system example, Jacobson and Ng have identified a number of functional aspects, such as *handle waiting list*, *checking in customer* and *handle no room*.

In the Theme approach by Clarke and Baniassad [8], a *theme* encapsulates a piece of functionality or aspect or concern that is of interest to a developer. At the requirements analysis level, themes are classified sets of requirements (taken directly from the requirements description document). Aspect themes are those that might be triggered in multiple different situations. They identify several examples of theme aspects, many of them being functional, e.g., functional crosscutting themes in a crystal collection game, namely, *Track-Energy*, *Challenge*, *Drop*.

In [26], Moreira et al use aspects to modularise and compose volatile concerns. Many of these volatile concerns are functional, such as *card solvency* and *calculate fares* in a subway system and *bidding*, *order handling*, *payment* and *monitoring* in a transport system.

D'Hondt and Jonckers [10] provide an approach for representing business rules as aspects. Business rules are highly domain- and application-dependent and crosscut other domain elements. Examples of such business rules include: *loyal customers are entitled to a 5% discount; all customers who have a charge card are loyal,* and so on.

3.3. On the Notion of Non-Functional Requirements being Solution Domain Properties

Steimann argues that non-functional requirements are not elements of the problem domain and are, instead, technical properties and therefore only appear at the solution domain level. This is not so, however. Several other well-established approaches (goal- and agent-oriented [7, 11], for example) have demonstrated the need for putting non-functional requirements at the forefront of developers' thinking. In fact, many of these properties reflect real stakeholder concerns, even at the strategic organisational level, and their existence can be noticed explicitly and implicitly in the requirements descriptions. Therefore, we should not put those concerns on hold until the implementation phase is reached. And, as mentioned earlier, if we are to prove that aspects exist at the modelling analysis level, we cannot ignore a significant part of what constitutes one of the primary bases for our work: the requirements descriptions.

Our auction system model in Figure 3 also shows a number of non-functional aspects, i.e., *security*, *concurrency*, *transaction*, *logging* and *availability*. Again, though these non-functional aspects will be present in other domains, the requirements pertaining to these will nevertheless be domain specific and dictate different types of needs. For instance, in the auction system, the security needs are mainly concerned with ensuring that all users accessing the system are authorised, the communication between the client and server uses a secure connection and so on. On the other hand, security requirements for a home security monitoring system will include ensuring that all doors and windows have locks, alarms are wired to those locks, motion detectors fitted, etc. Security requirements for a transportation system might be related to special arrangements necessary when transporting military assets or sensitive documents. Though the non-functional property *security* appears in all these domains, the specific requirements differ and so will the solutions to satisfy those requirements. Also note that the *transaction* aspect is also a property of the domain as it relates to customers completing their transactions and obtaining their goods. Just because it may map on to a concrete transaction processing aspect in the implementation does not imply that it is a property of the programming (as stated by Steimann). When analysing the problem domain, the concept of a transaction will have specific properties, e.g., a long transaction in an auction system where a customer places several bids on the same item in response to increasing bids from other users. This is in contrast to a transaction in a banking system where the general nature of a transaction is typically short: users go to the ATM or bank clerk to withdraw cash and the transactions do not last for days or weeks as is the case for an auction system. At the domain analysis level we are interested in modelling the semantics of a transaction from a user/stakeholder perspective and not from the perspective of specific locking or concurrency protocols that may be employed during implementation.

3.4 On the First-Order Nature of Aspects

The discussion in Sections 3.1-3 clearly demonstrates that functional and non-functional aspects are properties of the domain and therefore must be modelled at the same level of abstraction as other domain concepts being analysed. Here we offer further evidence of the first-order nature of aspects.

In his paper, Steimann offers a semi-formal proof regarding the second-order nature of aspects. This proof is founded on the presence of a base decomposition, i.e. he envisages that there will always be a dominant decomposition paradigm employed for modelling domain concepts and that aspects will crosscut concerns in this dominant decomposition. However, a number of approaches in AOSD remove the strong distinction between base concerns and aspects. Instead they take a multi-dimensional perspective on separation of concerns and their subsequent modelling. This means that there is no dominant decomposition. All concerns, whether they are functional or non-functional, classes or aspects, are at the same level of abstraction. This has significant advantages for domain analysis and modelling. One can *fold* or *project* one set of concerns on another set of concerns, as needed, to understand their mutual dependencies and influences, including crosscutting ones. This provides a powerful composition mechanism as all concerns are composable in a uniform fashion. Concerns can be incrementally composed to build composite concerns which can in turn be composed together to form more coarse-grained concerns. Such multi-dimensional approaches have been proposed for requirements analysis [8, 27, 37, 38], design [3, 8, 19] and implementation [3, 38]. Models in such approaches invalidate Steimann's proof as all concerns in a multi-dimensional model are first-order entities.

4. Quantification and Obliviousness

In [14] Filman and Friedman proposed a simple classification of the relationship between aspects and classes based on the notions of *quantification* and *obliviousness*. Quantification is defined as the ability of an AOP pointcut language to specify a predicate which can match a variety of join points in the static class definitions and dynamic object interaction graphs. Obliviousness, on the other hand, is the ability of a class to be *aspectised* without having to specially provide any hooks to expose the various join points that aspects might want to quantify over. The statement in [14] is, however, a position statement and the authors do not imply that their classification is the only classification of fundamental properties of AOP. Nor is the classification intended as a definition of the fundamentals of AOP. There are other classifications that focus on other facets of the relationship between aspects and classes. For instance, Kersten and Murphy [21] have proposed a classification based on their experience in developing the ATLAS web-based learning system. They categorise aspect-class relationships into:

- *class directional:* the aspects know about the classes but not vice versa. This is analogous to Filman and Friedman's obliviousness.
- *aspect directional:* the classes know about the aspects but not vice versa. This means that classes are no longer oblivious of the aspects. Classes may need to be

annotated to specify the intention of fields and methods, e.g., as in meta-data-based pointcut expressions [25], instead of relying on lexical matching in existing pointcut expression mechanisms.

- *open:* this is a union of aspect directional and class directional – both aspects and classes know about each other.
- *closed:* neither the aspects nor the classes know about each other. This applies to systems with strong encapsulation, e.g., [30].

Kersten and Murphy's classification demonstrates that there are several *non-oblivious* modalities of the aspect-class relationship. In fact, a number of application studies have shown that, in a variety of cases, obliviousness is neither achievable nor desirable. Kienzle and Guerraoui [24] and Fabry [12] argue that when modularising transaction management concerns only *syntactic obliviousness* is achievable, i.e. syntactic representation of aspects and class models may not contain direct references to each other. However, *semantic obliviousness* is not desirable as objects need to be aware of their transactional nature. Similarly, Rashid and Chitchyan [32] demonstrate that, in the context of a database application, persistence can be effectively aspectised. However, only partial obliviousness is desirable. This is because persistence has to be accounted for as an architectural decision during the design of data-consumer components – GUI components, for instance, need to be aware of large volumes of data so that they may be presented to users in manageable chunks. Furthermore, designers of such components also need to consider the declarative nature of retrieval mechanisms supported by most database systems. Similarly, deletion requires explicit attention during application design as mostly applications trigger such an operation.

Quantification too is only a desirable property of any AOSD technique. No doubt predicate-like pointcut expressions, e.g., [29, 36] provide a means to match a range of join points in design or code models. However, in several situations that Colyer et al. [9] refer to as *heterogeneous aspects*, a pointcut expression may only select a single join point (i.e. no pattern-matching a la AspectJ is employed). The encapsulation of a number of such pointcuts and their associated advice in an aspect still modularises a crosscutting concern, though pointcuts do not employ any quantification mechanism.

There are, of course, alternative aspect composition models that do not rely on predicate-like pointcut expressions. We discussed role-based composition in Section 2.1. Such role-based composition models are often found in aspect-oriented architecture design approaches where connectors and associated roles manage aspect composition [2, 30]. Similarly, the increasing drive towards semantics-based pointcut expressions in AOSD means that at first glance a pointcut may not be explicitly quantifying over multiple join points. However, the semantics to be matched by the pointcut expression will inevitably be implicitly quantifying over other system elements. One such semantics-based pointcut expression mechanism has been developed in the requirements description language from AOSD-Europe [6]. The language enriches existing requirements descriptions with additional semantics derived from the semantics of the natural language itself. Therefore, as shown in Figure 4, the constraint specification (analogous to a pointcut expression) can match all the aspect requirements where the subject of the sentence (in a grammatical sense) is a *seller* and the object (again in a grammatical sense) an *auction* with an *end* relationship between the subject and object. Similar semantics-based matching is done in the *base* and *outcome* expressions. Instead of using a syntactical match as in Figure

2 (bidding composition), we are instead matching elements based on the semantics derived from the requirements descriptions, i.e. the subject, object and nature of relationship between the subject and object. Such semantics-based join point models have also been proposed for aspect-oriented design [36] and programming [29, 33].

```
<Composition name="CancelAuction">
  <Constraint operator="begin/end">subject="seller" and relationship="end" and object="auction"</Constraint>
  <Base operator="ifNot">subject="auction" and relationship="begin"</Base>
  <Outcome operator="satisfy">all requirements where subject="start date“ or object=“start date”</Outcome>
</Composition>
```

Figure 4: Semantics-based composition in the AOSD-Europe RDL

So if *quantification* and *obliviousness* are not fundamental characteristics of an AOSD approach, what is fundamental for aspects? In our view, the same characteristics that hold for other separation of concerns mechanisms are also fundamental for aspects, i.e. *abstraction*, *modularity* and *composability*. It is not quantification and obliviousness but the *systematic* support for abstraction, modularity and composability of *crosscutting* concerns [34] that distinguishes AOSD techniques from other separation of concerns mechanisms.

4.1 Aspects are about Abstraction

Abstraction is a means to hide away the details of how a specific concept or feature may be implemented in a system. Abstract types provide us a means to reason about relevant properties of a problem domain without getting bogged down in implementation details. So the first question we need to address is whether aspects provide any benefits in terms of abstraction. In fact, abstraction is as fundamental to AOSD as it is to any other separation of concerns mechanism. The notion of an aspect allows us to abstract away from the details of how that aspect might be scattered and tangled with the functionality of other modules in the system. At the modelling level, aspects help us abstract away from implementation details, for instance, the examples of security and transactions in Section 3.3. At the same time, we can refine aspects at a higher-level of abstraction, e.g., aspects in requirements models, to more concrete aspects hence gaining invaluable knowledge about how crosscutting properties in requirements map to architecture-, design- and implementation-level aspects (this is discussed further in Section 5). This is analogous to refining objects in requirements models to their corresponding designs and implementations. The key difference is that, as abstractions, aspects facilitate tracing the impact and influence of crosscutting relationships through the various refinements.

4.2 Aspects are about Modularity

Abstraction and modularity are closely related. When we abstract away from specific details that may not be of interest at a certain level of abstraction, we also want to modularise details that are of interest so that we may reason about them in isolation. This is termed *modular reasoning* [23]. When modelling domain concepts, modular reasoning is fundamental to understand the main concerns of a problem and to reason

about the individual properties of the domain concerns. The modularisation of crosscutting requirements in aspects greatly facilitates such modular reasoning. Returning to our example from Section 3.2, the modularisation of bidding, selling and bid solvency requirements allows us to reason about the needs they impose on the system as well as about their completeness regardless of how they affect or influence various viewpoints in the system. The same applies to the non-functional aspects we discussed. Without modularisation of such crosscutting properties, we would need to reason about them by looking at their tangled representations in the various viewpoint requirements, which would be an arduous and time consuming task. Because these crosscutting concerns would be tangled with the viewpoints in the absence of aspect modularity, this is further evidence that they are properties of the domain and not of the programming solution.

4.3 Aspects are about Composability

Modularity must be complemented by composability. The various modules need to relate to each other in a systematic and coherent fashion so that one may reason about the global or emergent properties of the system – using the modular reasoning outcomes as a basis. We refer to this global reasoning as *compositional reasoning*. Aspects facilitate such compositional reasoning about the problem domain as well as the corresponding solution. For instance, when composing the various aspects and viewpoints in our auction system example, we can understand the trade-offs between the aspects even before the architecture is derived. For example, we can observe that the bidding and bid solvency concerns may be at odds at times: we wish to allow people to place bids yet the solvency requirements must prohibit this at times. This allows us to reason about the overall bidding process and its administration. In addition to reasoning about inter-aspect interactions, we can also reason about how aspects influence the requirements of the various viewpoints. For instance, the various viewpoints are constrained by the security requirements which may require customers to register, login and use secure transmissions before participating in any auctions. How this compositional reasoning is carried out is beside the point. Quantification in pointcut expressions is just one way of doing this. That does not mean that one is manipulating first-order elements in second-order expressions. The goal is to compose the various domain elements, i.e. the aspects, classes, etc. to be able to reason about the global properties of the system.

5. From Early Domain Aspects to Design and Implementation Aspects

Capturing aspects early in the life-cycle has several advantages. In particular, this can help to guarantee that all stakeholders' concerns are identified and captured properly, reducing the possibility of either losing significant requirements during development or else keeping them in a separate list that needs to accompany the developer through to the solution domain. Such an approach increases the consistency between

requirements, architecture, design and implementation, providing, at the same time, improved support for traceability of all types of concerns across the development lifecycle activities. Moreover, a systematic means to identify and manage crosscutting concerns at the problem domain level contributes to completeness of requirements specifications and their corresponding architecture, design and implementation. An evident consequence is that the requirements specification can truly function as a bridge to narrow the classic gap between the problem and the solution domains.

In [27] and [34] we observed that analysis of requirements-level aspects provides us with an improved understanding of their mutual trade-offs and, consequently, with the ability to make improved architectural choices. Each functional or non-functional aspect leads to a number of architecture choices that would serve its needs with varying levels of stakeholder satisfaction. These architectural choices are unlikely to be the same and could even be conflicting (which is often the case). All these, often conflicting architectural choices *pull* the final architecture choice in various directions. Our requirements-level trade-off analysis gives us some early insights into such a pull and helps us resolve some of the conflicts. This arms us with a better understanding of the diverse and conflicting needs of aspectual concerns hence facilitating the choice of an optimal architecture that balances these conflicting needs.

However, requirements-level aspects are more than just identifying architectural choices and trade-offs. A requirements-level aspect can be stepwise refined into one or more architectural aspects, and, subsequently, design and implementation aspects. For instance, in our auction system example, the bid solvency aspect would be refined into an aspect implementing specific algorithms for ensuring solvency across multiple, concurrent bids by the same customer. At the same time, such an aspect would require an awareness that customers could be selling items at the same time, and hence receiving top-ups on their accounts. The availability aspect, on the other hand, would map onto a decision for an architectural choice, i.e. involving backup servers, high stability network connections and so on. At the same time, it will also refine into concrete solution domain aspects realising replication, session management, etc.

Similar mappings have been proposed by others. Jacobson and Ng [17], for instance, handle each use case module, and in particular each use case slice, separately through architecture to code, by refining the analysis elements into design structures (classes and components) and, when necessary, adding new solution structures. In their examples, all the use case slices identified during the requirements analysis are kept during architecture and low level design. During architecture design, new aspects appear to keep platform specific elements separate from the platform independent ones. Similarly, in Theme [8], requirements analysis themes are carried forward to the design level – each analysis-level theme is designed separately from all the others and contains all the necessary solution domain structures to implement it.

6. Conclusion

In his conclusion Steimann encourages others to challenge and disprove his hypothesis and sets three conditions [35]:

1. *"The aspect must be an aspect in the aspect-oriented sense (in particular, it must not be a subroutine or a role);*
2. *It must not be an artefact of the (technical) solution, but must be seen as representative of an element of the underlying problem domain;*
3. *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."*

In this paper, we have shown several examples where aspects are not mere sub-routines or roles – i.e. they are first-class problem domain concepts that crosscut other problem domain concepts (satisfying condition 1). Modelling of such concerns as sub-routines would require them to be triggered by viewpoints in different situations, hence tangling these concerns with the core descriptions of the viewpoints. We have also shown that quantification and obliviousness, though desirable, are not fundamental properties of AOSD. However, even if these were to be considered fundamental, aspect-oriented requirements engineering approaches offer strong modularisation and composition mechanisms satisfying both obliviousness and quantification (in contrast to what Steimann affirms). We have demonstrated that functional and non-functional aspects represent important stakeholders concerns at the domain-level and therefore need a first-order representation (satisfying condition 2). Finally, we have pointed out a considerable number of arbitrary functional aspects that can be found in the existing Early Aspects body of work, therefore satisfying condition 3.

We hope to have convinced the reader that aspects are not about obliviousness and quantification, and that they represent important stakeholder concerns present in the requirements descriptions which cannot be ignored and left to be treated during the implementation phase. Aspects are about more fundamental software engineering principles. Aspects are about abstraction, modularity and composability. These are the lemmas that should guide our decisions throughout the development lifecycle.

Acknowledgements. This work is supported by the projects: AOSD-Europe (IST-2-004349), MULDRE (EPSRC EP/C003330/1) and SOFTAS (POSC/EIA/60189/2004). The authors wish to thank Ruzanna Chitchyan for helpful comments and discussions.

References

[1] AspectJ Project, http://www.eclipse.org/aspectj/, 2006.
[2] E. Baniassad, et al., "Discovering Early Aspects", IEEE Software, 23(1), pp. 61-69, 2006.
[3] D. Batory, et al., "Scaling Stepwise-Refinement", IEEE Trans. on Soft. Engg., 30(6), 2004.
[4] L. Bergmans, M. Aksit, "Composing Crosscutting Concerns using Composition Filters", CACM, 44(10), pp. 51-57, 2001.
[5] N. Cacho, et al., "Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming", Proc. AOSD Conf., 2006, ACM, pp. 109-121.
[6] R. Chitchyan, et al., "Initial Version of Aspect-Oriented Requirements Engineering Model", AOSD-Europe Report D36 (AOSD-Europe-ULANC-17) http://www.aosd-europe.net 2006.
[7] L. Chung, et al., Non-Functional Requirements in Software Engineering: Kluwer, 2000.
[8] S. Clarke, E. Baniassad, Aspect-Oriented Analysis and Design: The Theme Approach: Addison-Wesley, 2005.

[9] A. Colyer, et al., "On the Separation of Concerns in Program Families", Lancaster University Tech. Report COMP-001-2004 (http://www.comp.lancs.ac.uk/computing/aose).
[10] M. D'Hondt, V. Jonckers, "Hybrid Aspects for Weaving Object-Oriented Functionality and Rule-based Knowledge", Proc. AOSD Conf., 2004, ACM, pp. 132-140.
[11] A. Dardenne, et al., "Goal-directed Requirements Acquisition", Science of Computer Programming, 20, pp. 3-50, 1993.
[12] J. Fabry, "Modularizing Advanced Transaction Management - Tackling Tangled Aspect Code": PhD Thesis, Vrije Universiteit Brussel, Belgium, 2005.
[13] R. Filman, et al. (eds.), "Aspect-Oriented Software Development": Addison-Wesley, 2004.
[14] R. Filman, D. Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness", OOPSLA WS on Advanced Separation of Concerns, 2000.
[15] A. Garcia, et al., "Modularizing Design Patterns with Aspects: A Quantitative Study", Proc. AOSD Conf., 2005, ACM, pp. 3-14.
[16] J. Hannemann, G. Kiczales, "Design Pattern Implementation in Java and AspectJ", Proc. OOPSLA, 2002, ACM, pp. 161-173.
[17] I. Jacobson, P.-W. Ng, Aspect-Oriented Software Development with Use Cases: Addison-Wesley, 2004.
[18] JBoss Aspect Oriented Programming Webpage, http://www.jboss.org/products/aop, 2006.
[19] M. Kande, "A Concern-Oriented Approach to Software Architecture": PhD, EPFL, 2003.
[20] E. A. Kendall, "Role Model Designs and Implementations with Aspect-Oriented Programming", Proc. OOPSLA, 1999, ACM, pp. 353-369.
[21] M. A. Kersten, G. C. Murphy, "Atlas: A Case Study in Building a Web-based Learning Environment using Aspect-oriented Programming", Proc. OOPSLA, 1999, ACM, 340-352.
[22] G. Kiczales, et al., "Aspect-Oriented Programming", ECOOP 1997, Springer, pp. 220-242.
[23] G. Kiczales, M. Mezini, "Aspect-Oriented Programming and Modular Reasoning", Proc. ICSE, 2005, ACM, pp. 49-58.
[24] J. Kienzle, R. Guerraoui, "AOP: Does It Make Sense? The Case of Concurrency and Failures", Proc. ECOOP, 2002, Springer, pp. 37-61.
[25] R. Laddad, "AOP with Metadata: Principles and Patterns", Industry Talk at AOSD 2005.
[26] A. Moreira, et al., "Modeling Volatile Concerns as Aspects", Proc. CAiSE, 2006, Springer.
[27] A. Moreira, et al., "Multi-Dimensional Separation of Concerns in Requirements Engineering", Proc. Requirements Engineering Conf., 2005, IEEE CS, pp. 285-296.
[28] K. Ostermann, "CaesarJ", http://caesarj.org/, 2006.
[29] K. Ostermann, et al., "Expressive Pointcuts for Increased Modularity", Proc. ECOOP, 2005, Springer, pp. 214-240.
[30] M. Pinto, et al., "DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development", Proc. GPCE, 2003, Springer, pp. 118-137.
[31] A. Rashid, Aspect-Oriented Database Systems: Springer-Verlag, 2003.
[32] A. Rashid, R. Chitchyan, "Persistence as an Aspect", Proc. AOSD, 2003, ACM, 120-129.
[33] A. Rashid, N. Leidenfrost, "VEJAL: An Aspect Language for Versioned Type Evolution in Object Databases", AOSD 2006 Workshop on Linking Aspect Technology and Evolution.
[34] A. Rashid, et al., "Modularisation and Composition of Aspectual Requirements", Proc. AOSD Conf., 2003, ACM, pp. 11-20.
[35] F. Steimann, "Domain Models are Aspect Free", Proc. MODELS 2005, Springer, 171-185.
[36] D. Stein, et al., "Expressing Different Conceptual Models of Join Point Selections in Aspect-Oriented Design", Proc. AOSD Conf., 2006, ACM, pp. 15-26.
[37] S. M. Sutton, I. Rouvellou, "Modeling of Software Concerns in Cosmos", Proc. AOSD Conf., 2002, ACM, pp. 127-133.
[38] P. L. Tarr, et al., "N Degrees of Separation: Multi-Dimensional Separation of Concerns", Proc. ICSE, 1999, ACM, pp. 107-119.
[39] J. Whittle, J. Araujo, "Scenario Modelling with Aspects", IEE Proceedings - Software, 151(4), pp. 157-172, 2004.