

# Relating Architectural Views with Architectural Concerns

Nelis Boucké  
DistriNet, K.U.Leuven  
Celestijnenlaan 200A  
3001 Heverlee, Belgium

nelis.boucke@cs.kuleuven.be

Tom Holvoet  
DistriNet, K.U.Leuven  
Celestijnenlaan 200A  
3001 Heverlee, Belgium

tom.holvoet@cs.kuleuven.be

## ABSTRACT

Architectural views are at the foundation of software architecture and are used to describe the system from different perspectives. However, some architectural concerns crosscut the decomposition of the architecture in views. The drawbacks of crosscutting with respect to architectural views is similar to the drawbacks with respect to code, i.e. hampering reuse, maintenance and evolution. This paper investigates the relations between architectural concerns and views to identify why concerns tend to crosscut. We propose to extend the architectural description with slices and composition mechanisms to prevent this crosscutting and perform an initial exploration of these concepts in an Online Auction system. Within this limited setting the first results look promising to better separate concerns that otherwise would crosscut the views.

## 1. INTRODUCTION

Architectural design is generally considered as a crucial step to cope with the inherent difficulties of developing large-scale software systems. Software architecture is a key artifact since it embodies the gross-level structure and earliest design decisions that directly impact the expected quality attributes and subsequent design or implementation phases [7].

Aspect-Oriented Software Development (AOSD) enables modularization of crosscutting concerns within software systems [1]. Though originally related to the implementation stage, recently the support is extended to cover the whole of the software development cycle (known as Early Aspects (EA) [2]).

Dealing with concerns with high impact on the gross-level structure and quality attributes asks for an architecture-centric approach [8]. Such concerns are typically coarse grained and must be tackled on the architectural level, recognized by the EA report [26].

Our position is that separating concerns—that otherwise would crosscut the decomposition—must also be applied on design models. Architectural views are at the foundation of architectural description and decomposed this description in several design models. Each of this design models elucidates the system from a different viewpoints to emphasis certain facets. Unfortunately, some architectural concerns crosscut this views. The drawbacks of crosscutting with respect to architectural views is similar to the draw-

backs with respect to code, i.e. hampering reuse, maintenance and evolution. However, the relation between software architectural views and separating concerns with remains AOSD largely unexplored. This paper starts with investigation of the relationships between concerns and views to identify why some concerns tend to crosscut. We propose to extend the architectural description with slices and composition mechanisms to prevent this crosscutting and perform an initial exploration of these concepts in an Online Auction system. We believe that the main challenge for AOSD to software architecture lies in advanced composition mechanisms, similar to the extensions with additional composition mechanism in UML or source code.

**Overview** Section 2 introduces architecture, architectural concerns and the Online Auction system used as illustration. Section 3 investigates the relation between concerns and views and identifies crosscutting. Section 4 proposes an extension and illustration the Auction system. Section 5 discusses related work. Conclusions are drawn in Section 6.

## 2. ARCHITECTURAL CONCERNS

### 2.1 Software architecture

Software architectures covers the first design decisions to meet the essential quality requirements of the system. Often, the description is more abstract than UML class diagrams or sequence diagrams and will not necessarily have a one-to-one correspondence with code. A common definition for software architecture is [7]:

The *software architecture* of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

It is generally accepted that architectures are too complex to be described in a simple one-dimensional fashion and must be described using several views [13]. Each *architectural view* shows particular types of elements and the corresponding relationships between them. A *viewpoint* is a template from which to instantiate individual views [27]. Architectural views allow to separate concerns, since each view emphasizes certain facets of the solution as clear and concise as possible while deemphasizing and ignoring other facets. Many different viewpoints and set of viewpoints exist, amongst them [21, 13, 16]. Views can be categorized according to three dimensions [13]: (1) views about the structure of implementation units (module); (2) views about the runtime units or runtime behavior and interaction (component and connector, C&C); and finally (3) views about how the software relates to its deployment and execution environment (allocation). No fixed set of views is appropriate for every system, but broad guidelines advise to include at least one view from each of these dimensions.

## 2.2 Definition of architectural concern

*Concerns* in software systems are in essence ‘issues important for a stakeholder of the software system’. The software architect, being an important stakeholder and responsible for building the software architecture, will have his own set of essential issues that have a high impact on the software architecture. Such concern is typically more coarse grained than concerns handled on code level and it is essential that the software architect handles them well. *Architectural concerns* are defined as:

An architectural concern is a concern for the software architect with a high impact on the software architecture.

Software architecture is the first place where problem (requirements) and solution (architectural abstractions and domain knowledge) come together [25] and architectural concerns are influenced by both. Example architectural concerns could be coordination between entities, security or distribution. The next section contains examples of architectural concerns<sup>1</sup> in an Online Auction system.

We avoid the use of the term architectural aspect in this paper<sup>2</sup>. It is with no doubt intuitive by exploiting the analogy with implementation-level concepts of AspectJ. Still, the analogy might turn out to be problematic, it could be assumed as straightforward translation of the concepts in the AspectJ model [14]. Such direct translation is difficult since software architectures use completely different concepts. Additionally, the term ‘aspects’ does not cover symmetric approaches who use the same representation for both crosscutting and non-crosscutting concerns and deal with crosscutting in the composition specification.

## 2.3 Example: Online Auctions.

An Online auction system is a client-server system that supports auctions on the internet. User of the system are Sellers, Buyers and Maintainers. Sellers can get a list of current auctions, create new auctions and cancel running auctions. Buyers can get a list of the current auctions, join or leave an auction and may place bids. The auctions process is a simple Dutch Auction. Once the auction is started, Buyers can place bids on the product. Each bid must be higher than the previous bid. At the end of the auction, the current best bid for the item is accepted and Buyer and Seller are notified that the auction has completed and bank transactions (final Buyer is charged his last bid, Seller is charged for the transaction costs). Maintainers can get lists of current auctions, current users and log files of progress of current and previous auctions and are able to intervene when needed. The system must be distributed (constraint) and be secure (quality attribute). The simple form of security tackled here is that users must be signed in before being able to participate in auctions. When creating an account, users must specify credit card information, username and password.

While building such a system the software architect must consider several important concerns. We focus on auctions, user accounts and distribution. Firstly, the architect must identify the main modules and process needed to list, create and run auctions (*Auctions concern*). Especially, the auction process and interaction between the users involved should be covered by the Auctions concern. This concern covers the ‘basic’ functionality (auctions) of the system and significantly influences the coarse grained structure of the program. Secondly, the architect must insure that appropriate security measures are taken. Here, a simple mechanism to ensure that a user is signed in is used as an example (*UserAccounts*

*concern*). The UserAccounts concern influences the security guarantees of the system and has an influence on other views of the system, which motivates considering it as concern. Finally, the architect must explicitly tackle the distribution of the system, since it will have a far reaching effect on the structure of the architecture (Distribution concern). For distribution, a client-server architectural model with a single server is used. Using such architectural model will have a significant influence on the performance of the system. Other examples of concerns, not further elaborated here, are Graphical User Interface (GUI) or Maintenance. How the concerns are identified is considered outside the scope of this paper.

It is important to realize that the Online Auction system is a relatively easy system, selected as example because the basic concepts of such system are well known and its (limited) complexity allows to describe it within the space of this paper. Yet, such example provides first insights and paves the way to more complex examples.

## 3. RELATING CONCERNS AND VIEWS

Dealing with concerns asks for an architecture-centric approach [8]. One of the key questions is how crosscutting relates to architectural views [2], however, the relation between concerns and views is largely unexplored. An important observation for design is that not only crosscutting with respect to ‘code’ becomes important, but also crosscutting with respect to ‘design models’. In this case, the design models are architectural views.

**The problem:** Architectural views are at the foundation of architectural descriptions and are a typical architectural decomposition introduced to handle complexity by improving SoC. Still, there are concerns that crosscut, often unrelated, architectural views. In the next two sections (3.1 and 3.2), crosscutting is illustrated together with a discussion why this crosscutting appears. Finally, section 3.3 discusses the drawbacks of crosscutting with respect to architectural views.

### 3.1 Structural mismatch between concerns and views

There exists a structural mismatch between concerns and architectural views, leading to scattering of concerns over architectural views. Two causes can be identified for this scattering. Firstly, views are structured according to the module, C&C and allocation dimensions (as outlined in section 2.1) and a view typically uses one type of element. Yet, concerns do not necessarily align with the dimensions and manifest themselves in several of these dimensions. This leads to scattering, since one concern must be described in several architectural views. Consider the UserAccount concern as an example, which manifests itself in several dimensions. For example, UserAccounts introduces an AccountManager to encapsulate the management of user accounts. Next to this, the concerns also covers the behavioral description of logging in and checking if the user is logged in using a process views (C&C dimension).

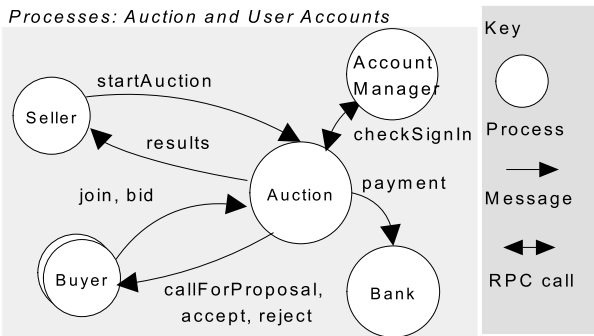
Secondly, elements of a single concern can be scattered over several views (even if they are from the same dimension) as a consequence of inherent complexity of large-scale systems. Consider the visual notation of processes as depicted in Figure 1 as an example (notation borrowed from [13]). If the view contains too many processes and relations, the whole view becomes cluttered. In the case of an Auction system the problem is less pressing, but for larger applications —with many processes— it is better to make several views focus on certain facets or a specific situation.

### 3.2 Limited composition specification leads to tangling

Tangling in architectural views occurs if a single view contains elements of several concerns. Which architectural elements are

<sup>1</sup>For the remainder of the text we use ‘concern’ when we mean ‘architectural concern’ unless otherwise stated.

<sup>2</sup>Originally coined by Tekinerdogan in [24], extensively discussed in [14]



**Figure 1: Process views illustrating background processes of the system.**

recorded in a single view is the software architect's decision. Thus tangling could be prevented by limiting architectural elements in a view to a single architectural concern. At first glance this seems to imply that tangling of concerns in views can be easily prevented by proper allocation of architectural elements to architectural views. A closer look reveals that much remains to be done before architectural views are really untangled.

There is no incentive to structure the architectural elements in views to better align with the concerns since concerns are not made explicit. And even with an incentive there remains a problem to be tackled. The problem lays mainly in the relationships between views or how the system can be composed from the separated views. One problem appears when a connector is needed between elements of a separate view (denoted as connector problem). The only way to connect the elements is by including them in the same view, i.e. in one of the original views or a in new view containing the elements and the connector between them. This obviously leads to tangling. For example, Figure 1 contains the process of both Auction and UserAccounts that can not be described without tangling because all processes are linked together with connectors. Apart from this, relations between architectural views are often implicit, e.g. elements having the same name. Other relations are informal, e.g. textual and informal descriptions of related view packages. Additionally, suggested by [13], the architectural documentation contains information beyond a single view and includes a mapping between elements in views (simple one-to-one or one-to-many mappings) and an element directory describing which element appears in which view. This type of relations suffices when considering classical use of architectural views. But additional and more powerful mechanism are needed when deepening the separation between views to better align with concerns. Without such mechanism, software architects are not encouraged and are not able to make untangled views.

### 3.3 Drawbacks of crosscutting

The drawbacks of crosscutting with respect to the architectural descriptions are similar to the drawbacks on the level of code. Firstly, since no single architectural view or set of views is identifiable as description of the concern, advantages of distinct design and development are lost. Before being able to update the design of a particular concern, an architect must review all views because there are few guidelines where to search. This clearly prevents traceability from architectural concerns to architectural elements and hampers maintenance and evolution of these concerns. Secondly, the standard notion of views does not allow explicit definitions of 'open spots' (abstract elements or parameters) that should be filled in later. Together with the traceability problem this hampers reuse

of architectural designs in other applications.

## 4. EXTENDING THE ARCHITECTURAL DESCRIPTION

Starting from the current notion of views and the problems identified in the previous section, we propose to extend the architectural description with slices and composition mechanisms. The proposal is initial in the sense that the concepts are only initially explored in a limited setting.

### 4.1 Extensions

Our first proposition is to extend the architectural description with the concept of an architectural slice (similar to hyperslices in HyperJ for implementation [23] or Themes for detailed design [5]). There are two types of slices: primitive and compound slices. Primitive slices are single views. Compound slices group several other slices (primitive or compound) together to cover a specific architectural concern. The advantages of architectural slices are twofold. Firstly, since the architectural elements in a slice are meant to cover a specific concern, there is a direct traceability between architectural concerns and the views describing them (and thus no tangling). For compound slices, the concerns are still scattered over several sub-slices, but they are clearly grouped together and directly traceable form the concerns. Secondly, hierarchical composition allows to gradually buildup the design of a large scale system. Such hierarchical composition can be very beneficial for the scalability of the approach.

Our second proposition is to allow architectural slices to have 'open spots' under the form of abstract elements (like UML or in [17]) or parameters (like in Theme/UML). These parameters can be bound to concrete values of other slices when describing the composition. Note that a single parameter can be bound to several concrete value, which is central to AOSD, i.e. to apply one concept at different places simultaneously. Introducing such 'open spots' is needed to cope with the reusability issues outlined in the previous section. Some of these parameterized slices could be seen as architectural patterns [9]. Parameterization will also help to tackle the connector problem of section 3.2 by defining a connector on a parameterized element and later on bind this element to the concrete element of another slice.

Finally, our third proposition is to extend the software architectural description to include an explicit composition specification for the slices. When describing the composition, slices are handled as first class elements and a connector encapsulates the composition specification. This extension is needed to allow the two previous extensions. As motivate in the previous section, more powerful composition mechanism are needed when deepening the separation of slices. Secondly, the parameters of a slice must be bound to concrete elements of other slices.

### 4.2 Illustration in Online Auction system

The concepts of slice, parametrization and composition specification are illustrated using a partial architectural description of the Online Auction system. The description involves three concerns: Auction, UserAccount and Distribution. All three concerns are tackled by a separate architectural slice, the UserAccount and Distribution slice involve parametrization and the composition is specified using composition diagrams. Not all details of all slices are presented through the text, the focus is rather on illustrating the extensions.

#### 4.2.1 Slices and parametrization

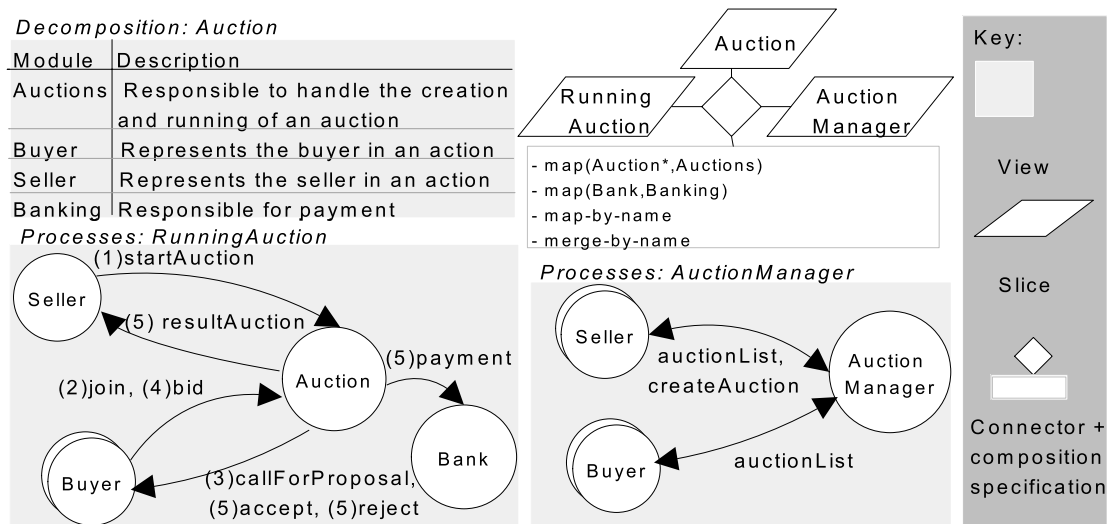


Figure 2: The Auction slice.

Consider<sup>3</sup> Figure 2 describing the Auction slice, containing three views and a composition diagram. The Auction slice could be considered as the 'base' slice because it covers the primary functionality of Online Auction system. The slice describes listing and creations of auctions, and auction process itself. The views are: (1) *Auction*, a module decomposition view identifying the static modules for the auction and their responsibilities. (2) *RunningAuction*, a process view describing the process and the interaction between those processes involved in a running auction. The numbers before the calls or messages indicate the sequence followed (these numbers could be omitted). If several calls or messages have the same sequence number, the order between them does not matter. (3) *AuctionManager*, a process view describing the process involved when creating and listing auctions. The composition diagram is explained further in this section.

The slice on User Accounts contains three views. Especially the view *CheckSignIn* is of interest because it introduces two new concepts: parametrization and interaction refinement. Parametrization is indicated by underlining the names of processes, calls or messages. The forked arrow indicates interaction refinement is inspired by [4] (see related work). Above the arrow is the description of the original interaction, below the arrow is the refinement of this interaction. The description of the original situation together with possible binding rules corresponding to a pointcut. The refined interaction can roughly be seen as the advice. Thus, the *CheckSignIn* view describes that before allowing any message from the user process to process X, process X will check that the user is properly logged in.

#### 4.2.2 Composition diagrams

The composition between slices are described in composition diagrams. Slices are represented by parallelograms, relations between slices by lines leading to a diamond. Such relations always contains a composition specification based on composition rules. The composition rules specify how the slices should be connected with each other. When reifying the composition, rules are applied from top to bottom.

During the composition specifications of the Online Auction system several composition rules have been used. Below is a list ex-

plaining each rule and its implications. But first, *expr* indicates an expression build up with text and wildcards (indicated by an asterisk). Brackets indicate that there that several expressions are listed. E.g. 'auction\*' will match everything starting with the text 'auction', '{auction\*, \*auction}' will match both elements starting or ending with 'auction'. *element* indicates that only a single element can be filled in.

- *match(expr, element)* : used to resolve naming differences between elements in different slices. Matching elements is only possible between elements of the same type (e.g. match process with process or module with module). An example can be found in Figure 4 where Seller and Buyer are matched with User.
- *map(expr, element)* : maps (multiple) element(s) on a single element. This composition rule can describe the same type of relations as the 'Mapping between views' tables of [13]. For example in Figure 2 this rule is used to map the Auction and Auction-Manager process (who both match Auction\*) onto the Auctions module. *map-by-name* will do a standard mapping between the elements based on names. As second example, consider Figure 5 where modules are mapped on other modules (thus forming sub modules).
- *generalization(expr, element)* : generalization relations between the elements who match the expression and the element in the right hand side of the relation. In our example, the composition rule is used to specify that user is a generalization of seller and buyer. In Figure 4, a visual notation is used to describe that user is a generalization of seller and buyer.
- *bind(expr, parameter)* : binds elements to the specific parameter. For example, in Figure 4 this relation is used to bind 'Auction' and 'AuctionManager' to X. Process X is a process for which the user must be logged in before it can be accessed.
- *merge(expr, expr)* : merges elements of the first expression with elements of the second expression if they are from the same type, similar to the merge operation defined for Theme in [11]. *merge-by-name* will merge elements on their names.

If no composition rules are specified, *map-by-name* together with *merge-by-name* are assumed. *match-by-name* is always automatically used, manual matching will only add additional matches. If some parameters are left unbound, they are just taken to be parameters of the the new composed slice (not used here).

<sup>3</sup>Graphical elements of figures are only defined once in a key. Also consult keys of previous figures and the text explaining the figure.

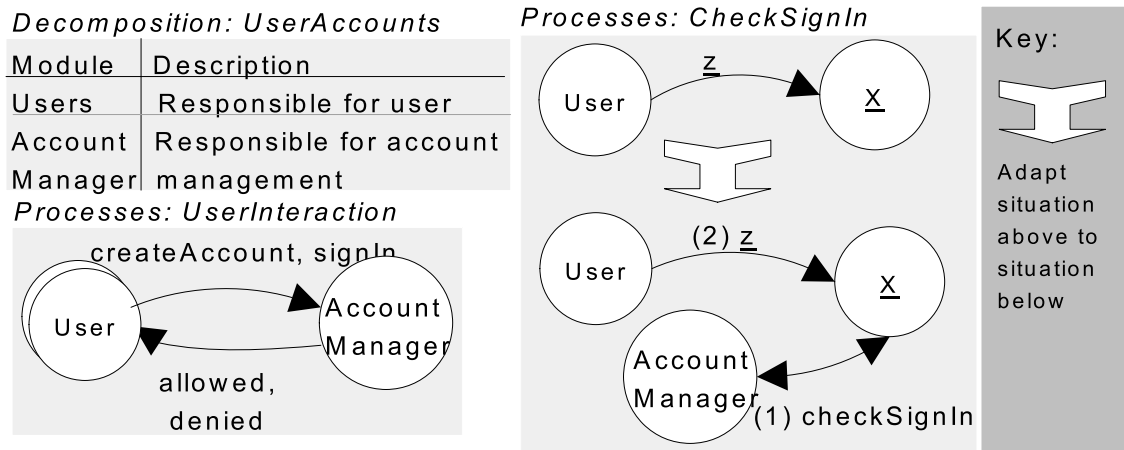


Figure 3: The UserAccount slice.

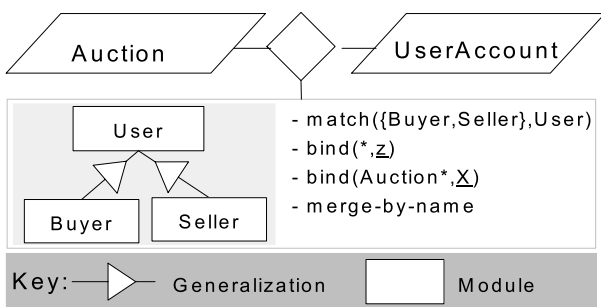


Figure 4: The AuctionAccount slice, merging the slices Auction and UserAccount together.

No difference is made between the reconciliation (resolve compatibility problems between different slices) and composition of the concerns, reconciliation can easily be defined as composition rules (e.g. match).

#### 4.2.3 Distribution influences both Auction and UserAccounts

As additional example, consider Figure 6. We elaborate on distribution because it provides a good example of a compound slice (distribution) that influences two other compound slices (Auction and UserAccounts, merged in the AuctionAccount slice).

The Distribution slice has four subviews. The *Subsystem* decomposes the system in a client, server and communication subsystems. Both the client and server can use the communication subsystem, described in the usage view *Communication*. *ClientServer* shows how client, server and communication are deployed on several computers connected by a TCP/IP network.

Finally, the combined view *RemoteMessages* shows that the connector between process A and process B (sending messages) must be refined and that messages must be sent through the communication subsystem. This communication subsystem ensures that the messages are sent through the TCP/IP network (e.g. taking care appropriate message format, appropriate host addresses, compression, etc.). Putting the process in the module in a combined view is equivalent to mapping processes on modules.

Figure 5 shows how the distribution slice is combined with the AuctionAccount slice. Care should be taken when interpreting the composition specification. The two first rules do map the appropriate modules onto client and server (forming submodules). This also

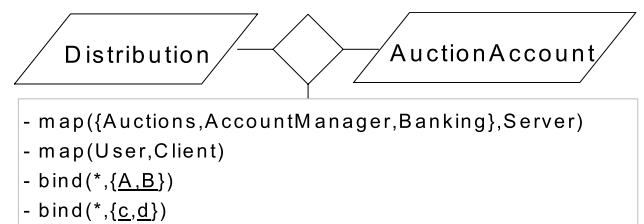


Figure 5: Composition of AuctionAccount slice and Distribution slice.

implies that all process that previously have been mapped on these submodules, now also belong to this module. Thus the processes of client are Seller and Buyer (and the generalization in User); the processes of server are Auction, AuctionManager, Bank and AccountManager. The next two rules might suggest to map any process on A and B and any message on c and d, but the constraint that A belongs to the Client subsystem and B belongs to the Server subsystem from the *RemoteMessages* view should also be taken into account. Constraints imposed by the view are applied before the composition rules. Thus Seller and Buyer (and their generalization in User) are bound to A and Auction, AuctionManager, Bank and AccountManager are bound to B. Bindings for c and d follow a similar approach.

### 4.3 Discussion

The quantification, an essential property of AOSD [15], over architectural elements is found in the expressions used in the composition rules. In the example, the expressions are rather limited (text and wildcards), but they can easily be extended. An important observation during the exploration is that rules tend to have different influences on different types of elements E.g. the mapping rule can be used to map process on modules, modules on computer systems and modules on modules (defining them as being submodules). Notice interaction refinement is currently used *within* the slice and not in the composition specification, although we think it is possible. We decided to include it in the view because this visual notation is very intuitive. Also, only a limited set of rules have been applied. Interesting possibilities to consider in the future are override composition operator (used in Theme/UML) and considering interaction refinement (used in Architectural Stratification, see related work) as a composition rule. A final remark on composition rules is that they strongly depend on the constraints imposed by the views

(as illustrated with the composition rules of Figure 5) and that they may interfere with each other. Currently, rules are only informally defined and it is clear that further research is needed to fully understand the possibilities and implications of using such rules to compose slices together. E.g. apply richer sets of composition rules; studying richer expressions in the rule parameters; formalizing the composition rules, their implications on the architectural elements and the process of applying the rules on the description language.

One of the main innovations is to make the composition diagrams part of the architectural description. Each composition diagram specifies a new compound slice, which could be reified by employing the composition specification. The architect could decide to reify a compound view to understand it better, or to incorporate it in the documentation for clarity reasons. Appropriate tool support can make such reification easier.

Until now, we used only a single connector per connection diagram. For example, we first composed Auctions and UserAccounts in a slice called AuctionsAccounts and then composed it with the Distribution slice. This is not strictly necessarily, we could compose the system together with two connectors in one diagram. Notice that the scope of a composition rule is limited to the slices linked with the connector in which the rule is specified. Composing all three slices in one diagram would keep the both connectors and composition rules of Figure 4 and Figure 5, the only change is that the latter must be reconnected to Auction and UserAccount instead of AuctionAccount. Further research is needed to understand the full implications of using multiple connectors in a composition diagram.

Mapping to previous and following development phases is not explicitly considered in this paper. Investigating the exact relations with concerns identified in requirements and concerns tackled during detailed design remains to be done.

## 5. RELATED WORK

A good survey on design approaches (both architectural and detailed design) can be found in [10]. Here, we only focus on approaches that are closely related to our work.

Rozanski and Woods [22, 28] identify that quality properties (for example security) appear in several architectural views. This is clearly related with the identification of crosscutting in the previous section. The authors introduce architectural perspectives as complementary to architectural views in the sense that they define a set of collection of activities, tactics and guidelines to ensure that the system exhibits a particular quality property. Architectural perspectives is not a technique for modular description but rather a framework to guide and formalize the process of ensuring that a particular architectural property is met, perspectives are *applied onto* views.

Atkinson and Kühne [4] propose architectural stratification to combines the strengths of component-based frameworks and model-driven architectures (MDA [3]) to support AOSD. The approach starts from the observation that you can define several structural architectures - depending on the level of abstraction at which you would like to see the system - each with different sets of connectors and components. The goal is to identify, elaborate and related the architecture on these different levels. The architectural description is structured according to architectural strata, which is a full description of the system on a specific specific level of abstraction (strongly related to architectural views). Architectural strata represent different level of architectural decomposition and lower strata refine the connectors of higher strata. The authors suggest that every stratum can be used to tackle a specific concern. Current limitations of the approach are that it supports only one type of relation (interaction refinement), the rigid structure of architectural

strata allowing only relations with the stratum above and below and limiting the description on a stratum to a single views. Our proposal, on the contrary, supports multiple types of relations, supports multiple views and has a less rigid structure.

In [19] Kande et al. study the need for multidimensional SoC in architecture descriptions. [18] proposes a concern-oriented framework called Perspectival Concern-Spaces (PCS). The goal is to develop architecture with as primary dimension the concerns, using an extension to UML as modeling language. But the PCS uses a very specific interpretation of IEEE-Std-1471 [27] (in which viewpoints are concern spaces) that differs substantially from the generally accepted interpretation of viewpoints in software architecture (e.g. described in [16, 22]). The interesting point raised by this work is relation between MDSOC [23] and architectural views.

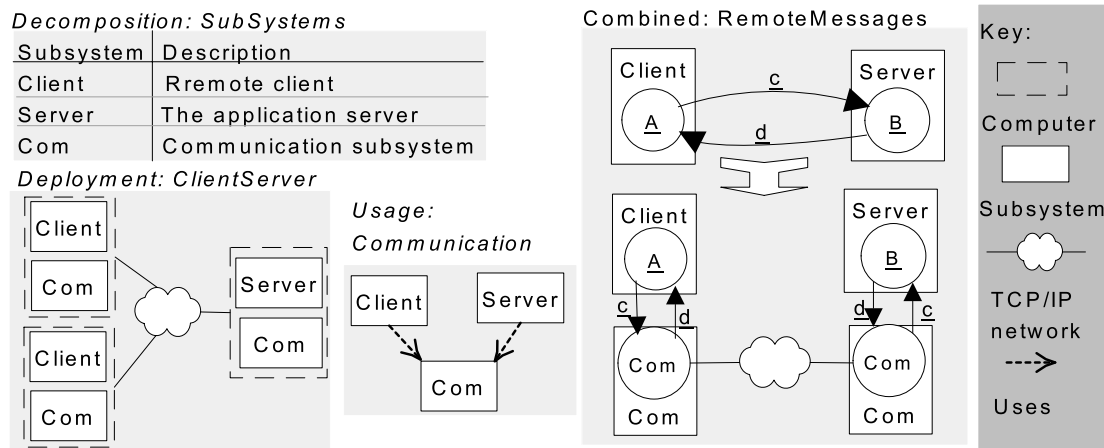
Theme [5, 11, 12] is an analysis and design approach to separate concerns proposed to cope with the structural mismatch between requirements and design using UML. For aspect oriented design, the approach defines a UML based aspect oriented design language called Theme/UML, extending the UML meta-model with model composition semantics. Concerns, crosscutting or not, can be separated in Themes (forming a symmetric approach). Every Theme contains several partial UML-models which are closely related to each other and can be parameterized. The most important contribution of Theme/UML is the explicit composition of Themes (UML-models) with composition operators like bind (binding concrete constructs to the parameters), merge and override. The differences with our approach is that Theme acts on detailed design level, e.g. on class diagrams and interaction diagrams and relying on all details of them. Secondly, Theme uses an explicit reconciliation phase while for our approach reconciliation is part of the composition rules. Finally, Theme uses only one composition rule to combine Themes while our approach allows specifying multiple rules.

Katara and Katz [20] observe that incremental design of aspects has been neglected and that cooperation or interference between aspects should be made clear at the design level. The authors propose an extension to UML and new architectural viewpoint (called concern diagram) describing how aspect can be combined to tread different concerns of a system. The concern contains dependency relations between aspects and shows which aspects contribute to which concerns. The work is related to our approach in the sense that relations between several aspects and concerns are made explicitly on the design level. Where Katara et al. mainly focusses on overlapping and interference between aspect in UML models, we focus on adding several types of non-trivial relations in the architectural description.

A recent paper of Baniassad et al. [6] also considers views as primary decomposition on the architectural level and identifies concerns crosscutting several views. The authors introduce the concept of an aspect view to capture concerns that otherwise would crosscut the decomposition in views. The difference with classical views is that aspects can contain abstract elements which is similar to the parametrization uses in our approach. The difference is in the concept of a slice as hierarchical composeable design building block and the explicit composition diagrams with connectors and composition rules used in our approach.

## 6. CONCLUSION

This paper investigates the relations between architectural concerns and views to identified that concerns tend to crosscut due to structural mismatch between concerns and views and limited support for composition between views. We propose to extend the architectural description with slices and composition mechanisms to prevent this crosscutting and perform an initial exploration of these



**Figure 6: The Distribution slice.**

concepts in an Online Auction system.

Within this limited setting the first results look promising to better separate concerns that otherwise would crosscut the views and to prevent the typical drawbacks of crosscutting. One of the first steps for future research is to more exactly define the composition process, the composition rules and their implications on architectural elements. Especially the composition rules ask for further research, e.g. apply richer sets of composition rules and study richer expressions. Additionally the implications of applying the extended architectural description to more complex case studies is an important challenge for future research.

## 7. REFERENCES

- [1] Aspect Oriented Software Development (AOSD) website. <http://www.aosd.net>.
- [2] Early aspects: Aspect-oriented requirements engineering and architecture design. <http://www.early-aspects.net/>.
- [3] Model driven architecture (mda) website. <http://www.omg.org/mda/>.
- [4] C. Atkinson and T. Kühne. Aspect-oriented development with stratified frameworks. *IEEE Software*, 20(1):81–89, 2003.
- [5] E. Baniassad and S. Clarke. Theme: An approach for aspect-oriented analysis and design. In *Proceedings of the International Conference on Software Engineering*, 2004.
- [6] E. Baniassad, P. C. Clements, J. Araujo, A. Moreira, A. Rashid, and B. Tekinerdogan. Discovering early aspects. *IEEE Software*, Januari/Februari, 2006.
- [7] L. Bass, P. Clements, and R. Kazman. *Software Architectures in Practice (Second Edition)*. Addison-Wesley, 2003.
- [8] N. Boucke and T. Holvoet. Dealing with concerns ask for an architecture-centric approach. In *Poster on the European Interactive Workshop*, 2005.
- [9] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, 1996.
- [10] R. Chitchyan, A. Rashid, P. Sawyer, J. Bakker, M. P. Alarcon, A. Garcia, B. Tekinerdogan, S. Clarke, and A. Jackson. Survey of aspect-oriented analysis and design, 2005. AOSD-Europe Deliverable No: AOSD-Europe-ULANC-9.
- [11] S. Clarke. Extending standard uml with model composition semantics. *Science of Computer Programming*, 44(1):71–100, 2002.
- [12] S. Clarke and R. J. Walker. Composition patterns: an approach to designing reusable aspects. In *Proceedings of the International Conference on Software Engineering*, pages 5–14, Washington, DC, USA, 2001. IEEE Computer Society.
- [13] P. Clements, F. Bachman, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures, Views and Beyond*. Addison Wesley, 2003.
- [14] C. E. Cuesta, M. del Pilar Romay, P. de la Fuente, and M. Barrio-Solorzano. Architectural aspects of architectural aspects. In *2nd European Workshop on Software Architecture (EWSA)*, volume LNCS 3527, pages 247–262, 2005.
- [15] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proceedings of the workshop on Advanced Separation of Concerns, OOPSLA*, 2000.
- [16] J. Garland and R. Anthony. *Large-Scale Software Architecture, A practical guide using UML*. Wiley, 2003.
- [17] S. Herrmann. Composable designs with UFA. In *Workshop on Aspect-Oriented Modeling with UML*, 2002.
- [18] M. Kandé. *A Concern-Oriented Approach to Software Architecture*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2003.
- [19] M. M. Kande and A. Stroheier. On the role of multi-dimensional separation of concerns in software architecture. In *Proceedings OOPSLA workshop on Advanced Separation of Concerns*, 2000.
- [20] M. Katara and S. Katz. Architectural views of aspects. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 1–10, New York, NY, USA, 2003. ACM Press.
- [21] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, November 1995.
- [22] N. Rozanski and E. Woods. *Software Systems Architecture*. Addison Wesley, 2005.
- [23] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. S. Jr. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.
- [24] B. Tekinerdogan. Asaam: Aspectual software architecture analysis method. In *Early Aspects 2003: Aspect-Oriented Requirements Engineering and Architecture Design*, 2002.
- [25] B. Tekinerdogan and M. Aksit. Synthesis based software architecture design. In M. Aksit, editor, *Software Architectures and Component Technology: The State of the Art in Research and Practice*, pages 143 – 173. Kluwer Academic Publishers, 2001.
- [26] B. Tekinerdogan, A. Moreira, J. Araujo, and P. Clements. Early aspects: aspect-oriented requirements engineering and architecture design, workshop report aosd2004, 2004.
- [27] The Institute of Electrical and Electronics Engineers (IEEE) Standard Board. Recommended practice for architectural description of software-intensive systems (ansi/ieee-std-1471), September 2000.
- [28] E. Woods and N. Rozanski. Using architectural perspectives. In *Proceedings of the WICSA conference*, 2005.