

Driving and Managing Architectural Decisions with Aspects

Alessandro Garcia¹

¹Lancaster University
Computing Department – InfoLab21
United Kingdom

{garciaa,marash}@comp.lancs.ac.uk

Thais Batista²

²Federal Univ. of Rio Grande do Norte
Computer Science Department
Brazil

thais@ufrnet.br

Awais Rashid¹

³Pontifical Catholic Univ. Rio Janeiro
Computer Science Department
Brazil

claudios@inf.puc-rio.br

ABSTRACT

Software architects face decisions every day which have a broadly-scoped impact on the software architecture. These decisions are the core of the architecting process as they typically have implications in a multitude of architectural elements and views. Without an explicit representation and management of those crucial choices, architects cannot properly communicate and reason about them and their crosscutting effects. The result is a number of architectural breakdowns, such as decreased evolvability, time-consuming trade-off analysis, and unmanageable traceability. Aspects are a natural way to capture widely-scoped architectural decisions and promote software architectures with superior modularity.

General Terms

Documentation, Design.

Keywords

architectural decisions, modularity, composability, aspect-oriented software development, early aspects.

1. INTRODUCTION

Software architecture is a fundamental element of modern software systems. Architects strive to develop adaptable architectures that are resilient in the face of changes especially for systems in volatile business domains such as eCommerce, banking, and telecommunications. In order to be adaptable, architectures must be modular. This serves a twofold purpose. If architectures are modular one can reason about individual architectural elements in isolation. This is termed *modular reasoning* [1]. At the same time, the various modules need to relate to each other in a systematic and coherent fashion to realize the intended architecture. Effective representation and specification of such relationships makes it possible to reason about the architecture as a whole – using the modular reasoning outcomes as a basis. We refer to this global reasoning as *compositional reasoning*.

Existing software architecture design and analysis approaches are geared towards supporting such modular and compositional reasoning. Architectural styles and patterns [9], for instance, are

based on the recognition of the effectiveness of specific organizational principles and structures. This helps one to undertake compositional reasoning about the elements deployed using a particular architectural pattern or style. Similarly, the notion of architectural components and connectors supports modular reasoning about individual architectural elements, i.e. the components, as well as compositional reasoning based on their relationships captured by the connectors. The Architecture Trade-off Analysis Method (ATAM) [3] supports modular reasoning by building and maintaining both quantitative and qualitative models of various competing quality attributes. These models are then composed to carry out compositional reasoning for identifying trade-off points. The “4+1” view model [5] separates an architecture into logical, process, physical, and development views, derived from the various stakeholders’ perspectives. This makes it possible for an architect to modularly reason about each of the views. A fifth view, the scenario or use case view, shows how elements in the other views work together thus supporting compositional reasoning.

Architectural decisions play a fundamental role in support of such modular and compositional reasoning as they are the driving force behind the architecture conception. They encompass critical architectural choices which have both structural and behavioral implications for the various architectural elements and the architecture we wish to reason about. It is, therefore, important to document architectural decisions in a systematic fashion. Several authors (e.g. [6, 11]) motivate the need for such documentation to support conveying of change, implications, rationale, and options as well as facilitate traceability and provide agile documentation. Our experience, however, shows that documenting architectural decisions alone is not sufficient. Architectural decisions have a broadly-scoped impact on the architecture.

Take, for instance, the “4+1” view of a software architecture that addresses several broadly-scoped properties, such as availability. When attempting to understand the availability-specific architectural decisions and their implications, an architect needs to reason across the various views, i.e. the logical, process, physical, and development views. This is because those availability decisions are likely to relate to multiple elements across more those views. This is particularly challenging as architectural decisions often lead to addition of new structure or behavior within a view. Since these implications are scattered across various view elements and views themselves, it is difficult to undertake modular reasoning about a particular decision. Compositional reasoning is even more challenging as one needs to understand the combined implications of various architectural decisions spanning a multitude of elements across several views.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SHARK '06, June 11, 2006, Torino, Italy.

Copyright 2006 ACM.

This implies that, in addition to systematic documentation of a decision, it is important to capture the additional behavior and structure it introduces into the various elements in the views. Furthermore, it is important to provide a composition mechanism that can quantify over the various elements in the views to compose such additional behavior and structure. This would support an architect to undertake modular reasoning about a decision and its implications. More importantly, by systematically exposing the semantics of a decision’s compositional relationship with architectural elements, we can support an architect to undertake compositional reasoning about the combined implications and trade-offs of various architectural decisions.

This paper presents our approach to providing such modular and compositional reasoning support, which is based on the use of aspect-oriented software development (AOSD) techniques [8, 10]. AOSD techniques provide additional support for modular and compositional reasoning by separating concerns that would otherwise be interspersed with other concerns in a software system. We can see from Figure 1a that AOSD techniques make it possible to modularize such concerns hence supporting modular reasoning about them; the colored dots represent join points used by the various aspects that have been modularized. At the same time, they provide a composition mechanism centered on the notion of join points, which are effectively a composition interface exposed on part of the non-aspectual elements of a system to facilitate aspect composition. In Figure 1a, the colored arrows are the composition specification using these join points. Hence, the notion of a join point model and the composition specification based on it facilitates compositional reasoning about broadly-scoped properties in a system. Since decisions have a broadly-scoped impact on the architecture they lend themselves as natural candidates to be aspectized.

This paper is organized as follows. Section 2 discusses the importance of coupling architectural decisions and aspects. Section 3 uses a case study to illustrate the crosscutting nature of architectural decisions. Section 4 contains a generic template that we propose to explicitly communicating crosscutting architectural decisions. Section 5 focuses on the composition of architectural decisions. Section 6 contains a discussion about the benefits of aspectizing architectural decisions and future trends in this context. Section 7 presents our final remarks.

2. ASPECTS DRIVING ARCHITECTURAL DECISIONS

Software is no longer engineered using a rigid separation of development stages. With the increasing adoption of iterative and agile methodologies, gone are the times when a strict separation between requirements engineering, architecture, design, implementation, and evolution was perceived as good practice. Key architectural decisions may be taken as early as requirements engineering. This is particularly true for COTS-based systems development where early risk analysis is important to understand the implications of using components with different communication mechanisms, data formats, etc.

Broadly-scoped concerns, whether functional or non-functional, e.g., availability, security, performance, informational retrieval, etc., identified during requirements engineering have important architectural implications. Aspect-oriented requirements engineering techniques [7] make it possible to systematically

identify, modularize, represent, and compose such broadly-scoped concerns. Such techniques, therefore, make it possible to modularly reason about such concerns as well as undertake compositional analysis for early identification of trade-offs among them. These broadly-scoped concerns and their mutual trade-offs provide early insights into the various architectural decisions facing an architect. As shown in Figure 1b, these concerns can be perceived to be various nodes of a lattice. Each concern leads to a set of architectural decisions. The mutual trade-offs exerted by the concerns, and the decisions driven by them, pull the architecture in various directions. The decisions need to be elaborated and represented in a modular fashion during architecture design. Effective modular and compositional reasoning about such architectural decisions is what helps the architect find the optimum point within the lattice where the final system architecture, satisfying the stakeholders’ concerns, would reside. For simplification, we have only highlighted three lattice nodes in Figure 1b. Note that the lattice can have as many number of nodes as the aspects found in the requirements specification.

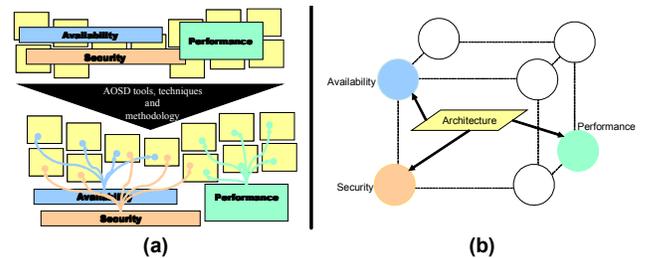


Figure 1: AOSD and Architectural Decisions (a) Modular and compositional reasoning about broadly-scoped properties using AOSD. (b) Architectural *pull* exerted by decisions pertaining to aspects.

3. CROSSCUTTING ARCHITECTURAL DECISIONS

Although the explicit handling and representation of architectural decisions are of paramount importance, they are not trivial tasks. Many decisions associated with relevant architectural concerns are crosscutting by their very nature and, as result, they need to be treated as such. They cut through the primary modularities of the architecture description, which is often consisted of one or more views. An architectural concern can affect several elements in an architecture description, such as components and their interfaces, relationships, processes, and also the decisions associated with other concerns.

In order to understand these problems, consider the architecture of a context-sensitive tour guide system. Figure 2 shows a partial description of the software architecture for this example based on a component-and-connector view [2] and on additional views from the “4+1” view model [5]. The upper left depicts a structural diagram with the component-and-connector view. The visitors use a *Navigator* to navigate through a tour, to create a customized tour, and to update information about the navigation preferences. The *Navigator* component contacts the *InformationRetrieval* component to recover information from the system. The *Navigator* also contacts the *ExternalServices* component to connect the visitor to external services. The *LocationManager* provides the identification of the current location of a visitor. This

identification is used by the *InformationRetrieval* component that provides tourist information according to his/her current location. The *TouristInfoManager* allows the tourist centre to update information in the system.

In this structural perspective of the tour guide architecture, it is also clear that the decisions with respect to the availability requirement affect several points of the architecture specification. Although availability-specific choices are somewhat localized in the *ReplicationManager* component, they largely impact on the definition of several interfaces and components, which do not have the primary purpose of addressing availability issues. Availability-related decisions crosscut multiple components, including *InformationRetrieval*, *LocationManager*, and *TouristInfoManager*. As availability support requires the replication of critical components and the replica consistency management, specific components and interfaces need to be created and added to those affected components. The crosscutting phenomenon also involves other concerns, such as security and performance.

The crosscutting manifestation leads to two major problems at the architectural level, the so-called *scattering* and *tangling*. Architectural scattering is the manifestation of architectural decisions, which belong to one specific concern, in several architectural units encapsulating architectural decisions referred to other architectural concerns. For example, the replication-related interfaces are scattered over multiple architectural components, such as *LocationManager*, *InformationRetrieval*, *TouristInfoManager* components (upper left of Figure 2). Architectural tangling is the mix of multiple concerns together in the same architectural elements. For instance, tangling is evident in the *InformationRetrieval* component since it is realizing an availability-related interface in addition to its primary functionality of providing information.

As previously mentioned, there are some architectural aspects which bring deeper problems to the software architects; they can even crosscut other architectural views in addition to the structural view, as it is the case for the availability concern. The availability-specific decisions are scattered and tangled within elements of other concerns over the four architectural views. Availability requires not only the inclusion of components, interfaces, and connectors (component-and-connector view), but also the definition of two separate threads to manage both replication and consistency (process view), the conception of the management layer together with other supplementary managers (development view), and the distribution of replication elements through different servers (physical view).

Traditional architectural approaches such as 4+1 view model [5], ATAM [3], tactics [2], and architectural styles or patterns [9] have different complementary purposes. However, they are not aimed at supporting the separate handling of crosscutting architectural decisions as exemplified in Figure 2. It brings in turn a number of substantial pitfalls, such as:

Hindering of modular and compositional reasoning. Tangling and scattering of decisions hinder both modular and compositional reasoning at the architectural stage. The architects are unable to reason about an architectural concern while looking only at its description, including its core decisions and structural and behavioral implications. Hence its analysis inevitably forces architects to consider all the architectural artifacts in an *ad hoc* manner. For example, the architects treating the availability and

security concerns in Figure 2 need to consult the definitions and decisions associated with all other architectural concerns across all the different views.

Traceability is unmanageable. Many of the concerns in the requirements specification entail crosscutting architectural decisions. The mapping of those concerns to the respective decisions is awkward as the developers do not have proper ways to easily check whether and how the requirements are met in the software architecture. For example, the association of availability-specific requirements with their architectural implications is cumbersome and far from being trivial. This obstacle makes it difficult to assess the goodness of the software architecture even in the presence of a good requirements engineering process.

Decreasing evolvability. Architecture degeneration is becoming very common in an age where software systems are always changing. Architecture artifacts are often key deliverables in the evolution process. As a consequence, the architects have additional work to answer recurring questions: What happens if we decide to change security-related components of our system? Has this decision been affected by which architectural concerns? As a complex architecture probably reflects thousands of crosscutting decisions, finding the answers for these questions is naturally time-consuming, especially when the original architects are no longer available.

Loosing essential information. With traditional approaches, software architects are not able to locally express the structural and behavioral implications of a given architectural decision in several architectural elements and views. The result is that important information is irrecoverable just because the lack of support for properly specifying them. Not only the final choices can be lost, but also the crosscutting rationale and competing options the architects considered.

Reducing reuse possibilities. Tangling and scattering are two of the main anti-reuse factors in the software lifecycle. The lack of a clear separation of concerns generates undesirable burdens on architectural reuse. For example, software architects may want to recycle, or at least remember, a comprehensive list of decisions and the rationale associated with an architectural concern in posterior projects. It would be certainly beneficial in order to empower software architects to reuse successful crosscutting architectural choices from previous projects.

4. CAPTURING ARCHITECTURAL DECISIONS AS ASPECTS

In the light of the mentioned problems, we conjecture that crosscutting architectural decisions should be handled as separate architectural aspects. The idea is to have proper abstractions to enable their representation as first-class elements, and also provide the means to facilitate their further composition. Aspects were originally conceived to address crosscutting concerns at the programming level [10]. It is then natural to believe that the key for capturing crosscutting architectural decisions is exploiting some AOSD concepts [8] at the architectural level.

Architectural aspects are units of modularity to capture the decisions associated with broadly-scoped concerns, letting the architects to represent all the structural and behavioral implications in a single place. Figure 3 shows templates to specify architectural aspects with essential information to capture crosscutting decisions: (i) *name* of the architectural aspect; (ii)

structural and behavioral architectural decisions, such as the inclusion of components, interfaces, relationships, processes, and so forth, which were made with the sole purpose of contemplating issues related to the architectural aspect; (iii) *composition rules* to describe how the crosscutting decisions with respect to this architectural aspect affects other architectural elements and alternatively other aspects; and (iv) a *reasoning section* that captures the rationale behind those decisions.

The crosscutting decisions affect several architecture elements, which are named *architectural joint points*. An architectural joint point is an element of interest in the architecture description through which two or more architectural decisions may be composed. Examples of joint points are: a component, an interface, a process, an architectural aspect, or even an architectural decision. *Architectural composition rules* support the composition specification and enable compositional reasoning. They are means of referring to collections of architectural joint points and describing some architectural decisions to be applied at those joint points.

Figure 3 shows how to use the notion of architectural aspects to support the modular description of the availability, security and performance concerns in our running example. All the availability-specific decisions are clearly captured in the first

template, including the creation of a *ReplicationManager* and system replicas, and the definition of two processes for controlling the system replicas and their global consistency. The rationale behind the availability decisions are reported in the reasoning section of the template. The reasons are related to structural and behavioral decisions as well as the composition decisions. In a similar way, the security-related and performance-related decisions are respectively isolated in the 2nd and 3rd templates.

As a result, the template-based specification is a cohesive manner to describe those broadly-influencing concerns which otherwise would be scattered and tangled over the architecture description and its multiple views. Notice that this approach is general and agnostic to different architectural representations that the software developers are relying on, whether graphical or textual, such as ADLs (Architectural Description Languages), UML-based or XML-based notations. The software architect can also use the templates in conjunction with multiple architectural views, and any existing notations for reflective design, where design rationale is extensively recorded [6]. In fact, the template can be used to describe all the kinds of architectural decisions and rationale, including assumptions, constraints, positions, arguments, status, and the like.

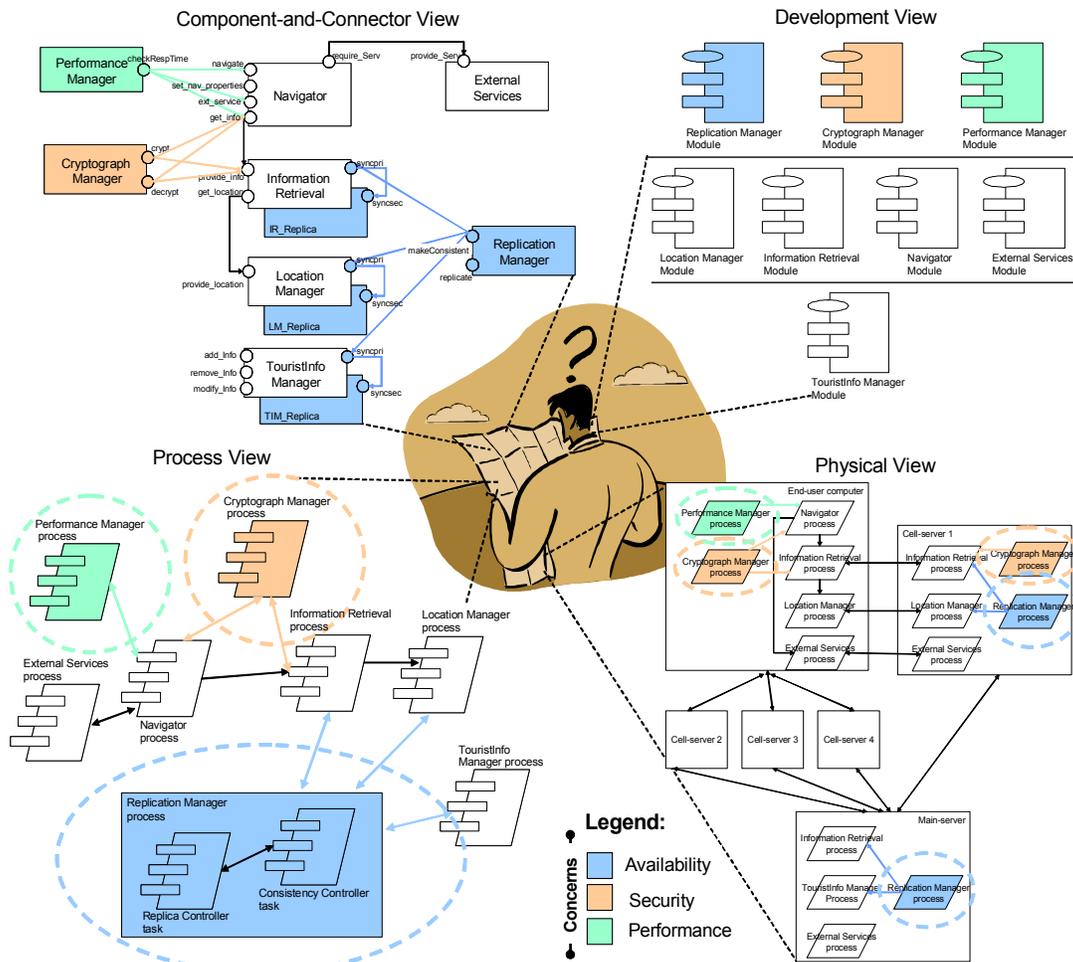


Figure 2: Tangling and Scattering in an Architecture Description

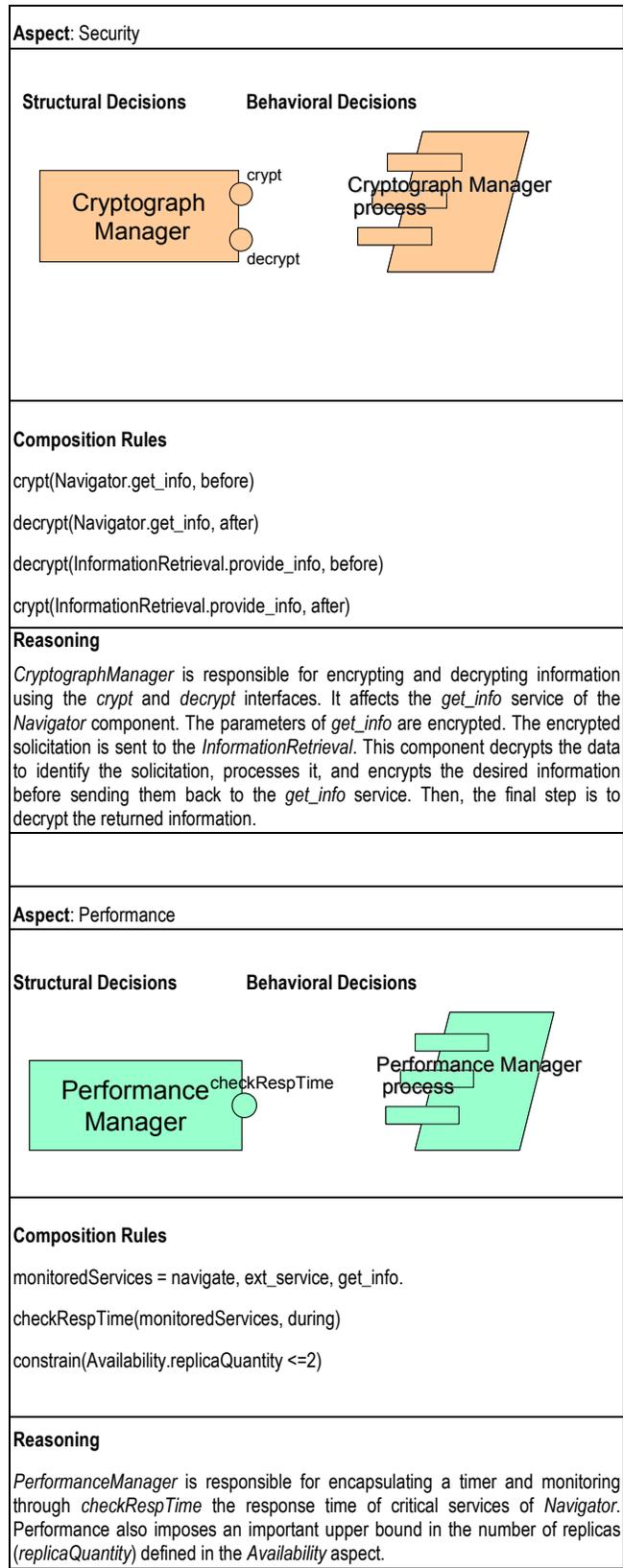
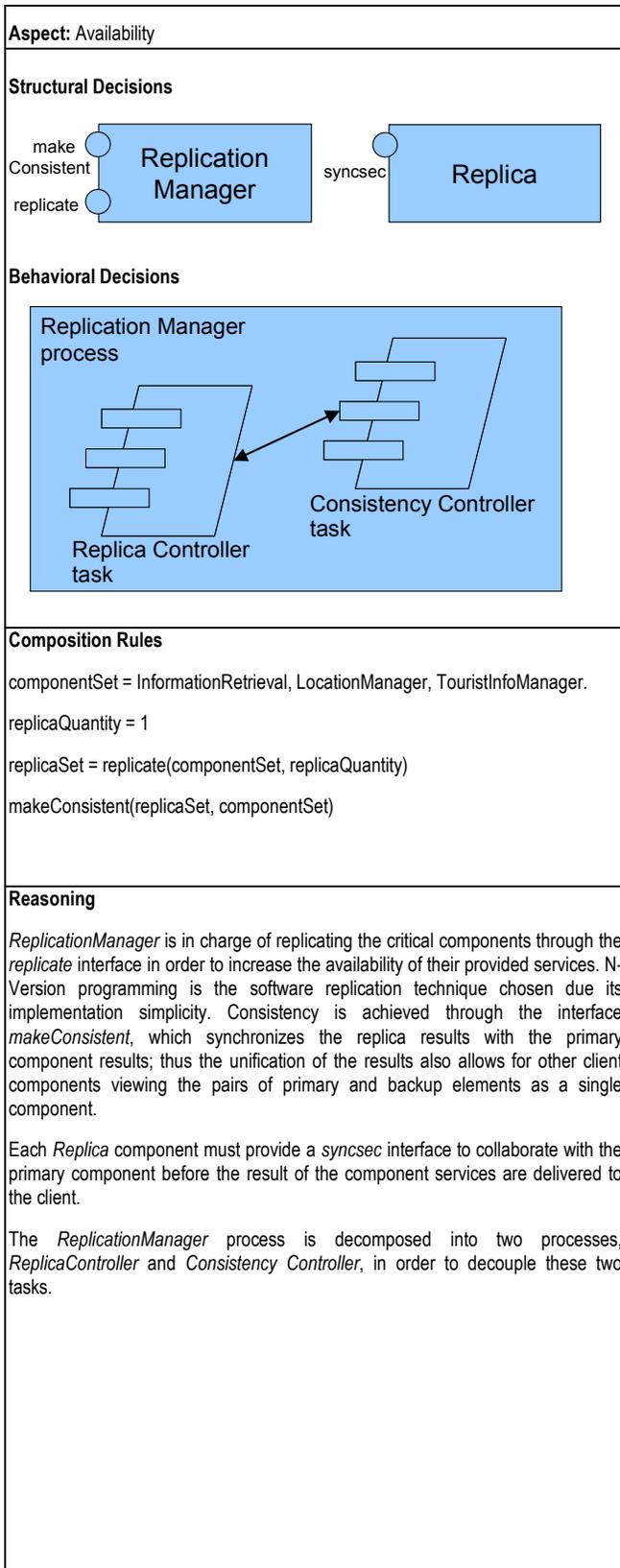


Figure 3. Modularizing and Composing Architectural Aspects

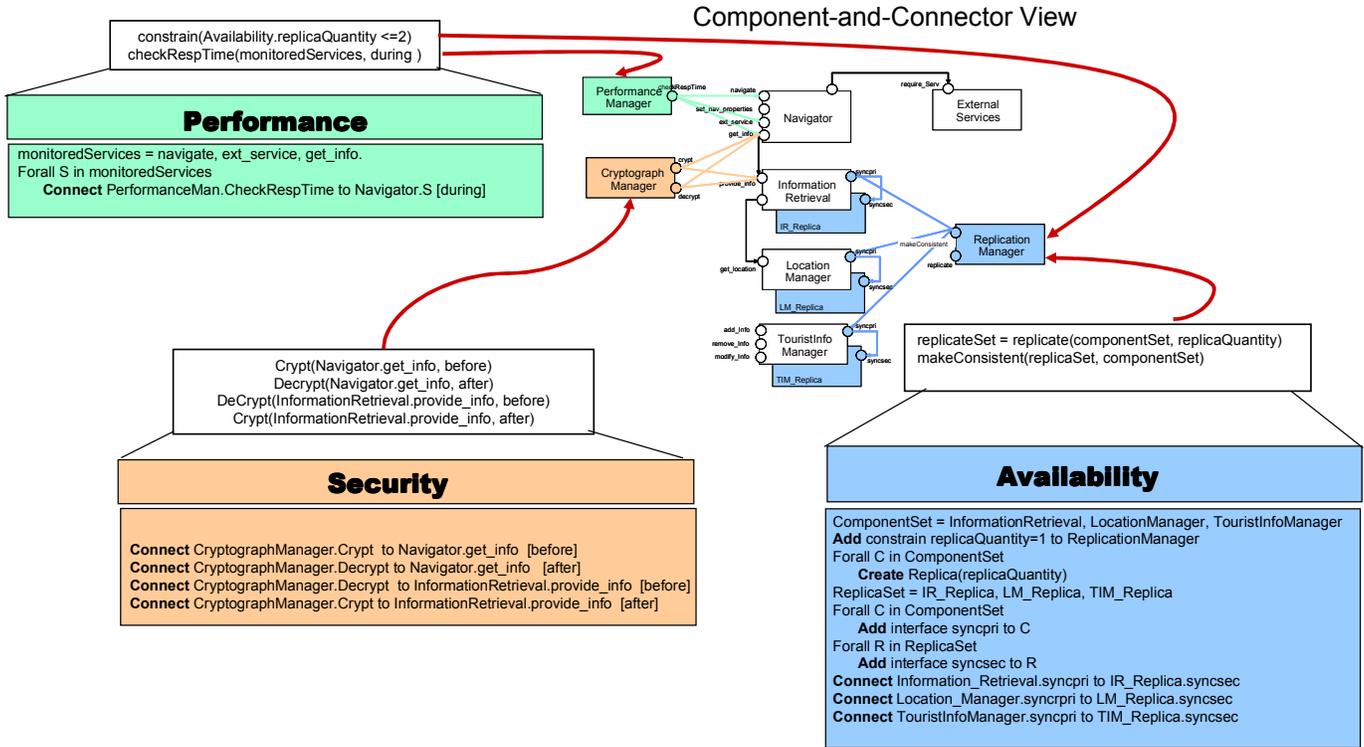


Figure 4. Mapping Architectural Aspects to Architectural Views: Effects of Architectural Aspects in the Component-and-Connector View

5. COMPOSING ARCHITECTURAL DECISIONS

Properly documenting the composition of architecture decisions is critical because architects make them in complex environments and they involve trade-offs. The architects can use a high-level composition language to facilitate the registration and communication of broadly-scoped choices and enhance compositional reasoning. Figure 3 shows how to work with a high-level language to describe those choices as architectural composition rules. The naming of the architectural decisions is intuitive as it actually captures the architectural operation associated with the crosscutting decisions.

For example, the third composition rule in the first template (Figure 3), named *replicate*, captures the fact that a list of architectural components should be replicated due to availability purposes. Auxiliary declarations can be made in order to facilitate the quantification process, such as the use of *componentSet* and *replicaSet*. The first rule uses *componentSet* to quantify the architectural join points affected by the *replicate* decision. Those points are critical components to be duplicated with different implementation versions, namely *InformationRetrieval*, *LocationManager*, and *TouristInfoManager*. The rule *makeConsistent* abstracts the process of including architectural elements to address the consistency of the primary components and their replicas.

To facilitate the composition of architectural decisions, the rules can pick out different types of architectural join points, such as interfaces or even rules defined in other architectural aspects. Figure 3 shows the *crypt* and *decrypt* decisions in the security

Table 1: Mapping rules

MAPPING RULE	DESCRIPTION
Add <elem_type> <elem_name1 [=value]> to <elem_name2>	introduces an architectural element of type <elem_type> and name <elem_name1>, optionally set its value, to other architectural element <elem_name2>
Modify <elem_name1> to <elem_name2 value>	changes the semantics of an architectural element by modifying its name from <elem_name1> to <elem_name2> or setting a new value to <elem_name1>
Remove <elem_type> <elem_name1> from <elem_name2>	removes an architectural element of type <elem_type> and name <elem_name1> from other architectural element <elem_name2>
Split <elem_type> <elem_name> into <elem_name_list>	separates an architectural element of type <elem_type> and name <elem_name> into two or more elements defined in <elem_name_list>
Unify <elem_type> <elem_name_list> into <elem_name>	groups two or more architectural elements defined in <elem_name_list> in the architectural element <elem_name>
Connect <elem_name1> to <elem_name2>	defines a relationship between the elements <elem_name1> and <elem_name2>
Disconnect <elem_name1> from <elem_name2>	removes a relationship between the elements <elem_name1> and <elem_name2>

aspect affect interfaces of *Navigator* and *InformationRetrieval*. The third rule of the performance aspect, named *constrain*, influences an availability rule that specifies the number of replicas. This rule represents a recurring scenario faced by software architects: several aspectual decisions affect each other. The aspect-oriented templates promote composition interfaces that allow for the architect to make it explicit the relationships and mutual influences of broadly-scoped concerns, which are not easily captured in traditional architectural views. In fact, this architectural constraint involving performance and availability components was not explicitly represented by any of the views in Figure 2. Some behavioral information can also be part of the composition rules. For instance, the specification of the security aspect also includes “when” the *crypt* and *decrypt* decisions should actuate over specific architectural elements, i.e. “before” and “after” requests of services of *Navigator* and *InformationRetrieval*.

As previously mentioned, architectural aspects can influence decisions made in several views. The architect may want now to review together the crosscutting decisions and the architectural views with the rest of the project team and the project stakeholders. Hence once the architectural aspects have been defined, the actual effect of the decisions in the multiple views may need to be specified and analyzed. The next alternative step then would be to use underlying composition mechanisms to support the mapping of aspectual decisions in terms of elements of the other architectural views. Those mechanisms can rely on mapping rules that simply translate the aspectual decisions in terms of the corresponding elements in the architectural views. Figure 4 shows how those mapping rules could be applied for mapping availability, security, and performance decisions to elements of a component-and-connector view. A similar mapping process could be carried out for the other architectural views. Table 1 shows a foundational set of mapping rules.

6. DISCUSSION AND RELATED WORK

This section discusses the benefits (Section 6.1) of modularly capturing crosscutting decisions using our approach described in Sections 4 and 5. This discussion is based on our extensive experience in both building aspect-oriented software architectures for different application domains [12-17], and defining and assessing aspect-oriented abstractions to the architectural stage [12, 18, 20]. Section 6.2 discusses related work while pointing out directions for future work.

6.1 What are the Benefits?

In the beginning, we identified numerous problems in conventional architecture-centric development approaches. By aspectizing crosscutting architectural decisions, we were able to address those issues and bring additional benefits:

Promoting modular and compositional reasoning of architectural decisions. With aspectual templates architects can reason about the otherwise crosscutting concerns in isolated and combined manners. In fact, the template sections describing the reasoning and the composition rules are more than just simple decisions – they also communicate the compositional rationale, and from where the structural and behavioral decisions came from.

Ease of traceability. Architectural aspectization lets you trace decisions back to concerns in requirements (such as, availability,

performance, and security). It also improves the identification of candidates to design and implementation aspects, linking them with their counterparts in the design and implementation artifacts. Moreover the composition rules inform the design team that those architectural elements are potential structures and behaviors to be modularized as design and implementation aspects.

Enhancing evolvability. In architectural evolution processes, the aspectual templates let architects by and large know the effects the previous design decisions had in the evolving system. Without such an explicit handling of architectural choices, the evolution process would likely lead to the violation of relevant crosscutting assumptions and influences that were not properly documented just because there was no proper support for their expression.

Promoting knowledge management and reuse. An aspect-oriented approach enriches the knowledge embedded in architectural models. We explicitly model the implications of broadly-scoped properties, in the same way we model components, interfaces, processes, or a design space of possible architectural solutions. This externalizes architectural knowledge present in a development team or organization, and is the basis for reuse.

Achieving Simplicity. Anybody can read the templates and respective composition rules in Figure 3, and understand how the team developed them. The architects do not need to change the way that they work while expressing architectural aspects. The aspectual templates can be seen as a complementary architectural view in addition to the views commonly used by the architects.

6.2 Where do We Go from Here?

Nowadays companies rely on architectural design reviews as critical points. Architects recognize the importance of making explicit assumptions [21] and tradeoffs within the architectural design space. However, the management of broadly-scoped architectural concerns is still made in an idiosyncratic fashion, with limited support for their modular and compositional reasoning. The next 10 years of research on software architecture will certainly have to face this problem. The marriage of software architecture and aspect-orientation is a key to address this challenge at different levels:

1. Identification and modeling of architectural aspects and their crosscutting decisions. The crosscutting nature of architectural decisions can manifest in several ways. As a result, architectural aspects require proper mechanisms and notations to identify, represent, and compose them. There are some modeling approaches to exclusively support the explicit description of architectural aspects at the logical view [4, 12, 18, 20]. To the best of our knowledge, there is no work in the literature that provides modularity mechanisms to capture crosscutting decisions, as we have presented in this paper.

2. An aspect-oriented architectural view. As the architecture of a system are represented by several views, each providing a distinct perspective of the system, the crosscutting concerns must be also modularly represented in the multi-view scenario. There is no research work that copes with an aspect-oriented architectural view and the provision of multi-view “weavers”, which automate their composition [20]. Such a view would simplify the architecting process and give a better picture of the system overall structure.

3. Aspectization of ADLs. There is a need for development of methodologies and tools to bridge the gap between the decisions specification and the ADL-based artifacts in order to maintain the integrity of architectural decisions. How to represent the architectural decisions at the ADL level? Although recently various proposals [4, 12] that integrate aspect-orientation and ADLs have been emerged, they do not cope with crosscutting architectural choices. Some works extend the component-connector abstraction to represent architectural aspects and composition rules as first-class elements. Others include this concept inside the component-connector abstraction.

4. Assessing Aspect-Oriented Software Architectures. It is almost always cost-effective to assess the crosscutting design choices as early as possible in the life cycle. Thus, to foster the benefits of more modular software architectures, we also need architecture design analysis methods to evaluate if the architecture reflects a proper modularization and composition of architectural aspects. Traditional methods for architecture assessment, such as ATAM [2], can be extended to deal with those issues. Tekinerdogan [22] provides a first step in that direction by defining an ATAM extension to support the identification of candidates for architectural aspects.

To address these challenges we can benefit from current notations, methodologies, languages, and tools and go a step further by adapting them to the new dimension of architecture design – the architectural crosscutting concerns and their composition. The adaptation of existing methodologies and tools avoid the need of the industry to deal with the burden of adopting new products in order to take advantage of the benefits of enhanced separation of concerns at the architectural level.

7. CONCLUSIONS

Architectural decisions are in the heart of the software development process because they provide the bridge between the problem space and the solution space. The promotion of modular and compositional reasoning about architectural decisions is essential to help software developers to understand if they got an architecture right according to their requirements. It is also a critical success factor for further system design and implementation. However, the broadly-scoped nature of early design choices imposes a number of problems to software engineers. In fact, architectural crosscutting concerns are even more challenging than implementation crosscutting concerns. While the latter typically impacts a single artifact (source code) often based on a single programming language, crosscutting concerns at the architectural level impacts a multitude of views with heterogeneous representations. Using only conventional approaches architects often get in trouble because important influences are scattered and tangled in the architectural views.

Based on our experience, AOSD techniques can certainly help organizations to improve their state of practice of software architecture. They support software architects with enhanced modular and compositional reasoning, which are imperative throughout all the software development phases. They also complement existing architecture-centric development approaches, both upstream and downstream. Upstream, aspect-oriented abstractions provide a natural way to modularize and compose decisions that are directly influenced by broadly-scoped concerns coming from the requirements. At the same time, downstream, explicit representation of architectural

aspects facilitates the satisfaction of top-level crosscutting decisions at the detailed design and implementation stages.

ACKNOWLEDGEMENTS

This work is supported by European Commission Grant IST-2-004349: European Network of Excellence on AOSD (AOSD-Europe).

8. REFERENCES

- [1] Kiczales, G.; Mezini, M. Aspect-Oriented Programming and Modular Reasoning. Proc. of ICSE'05, May 2005.
- [2] Bass, L.; Clements, P.; Kazman, R. Software Architecture in Practice. Addison Wesley, 2nd Edition, 2003.
- [3] Clements, P.; Kazman, R.; Klein, M. Evaluating Software Architectures: Methods and Case Studies. A. Wesley, 2002.
- [4] Cuesta, C.; et al. Architectural Aspects of Architectural Aspects. 2nd European Workshop on Software Architecture (EWSA), LNCS 3527, pp. 247-262, 2005.
- [5] Kruchten, P. Architectural Blueprints – The “4+1” View Model of Software Architecture. IEEE Software 12 (6), Nov 1995, pp.42-50.
- [6] Tyree, J.; Akerman, A. Architecture Decisions: Demystifying Architecture. IEEE Software, March/April 2005, pp. 19-27.
- [7] Rashid, A.; Moreira, A.; Araújo, J. Modularization and Composition of Aspectual Requirements. Proc. AOSD.03, pp. 11-20.
- [8] Filman, R.; et al. Aspect-Oriented Software Development. Addison-Wesley, 2004.
- [9] Shaw, M.; Garlan, D. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall. 1996
- [10] Kiczales, G.; et al. Aspect-Oriented Programming. In Proc. of ECOOP, pp. 220–242, 1997.
- [11] Louridas, P.; Loucopoulos, P. A Generic Model for Reflective Design. ACM TOSE), Vol. 9, Issue 2, April 2000, pp. 199 – 237.
- [12] Batista, T., Chavez, C., Garcia, A., Sant'Anna, C., Kulesza, U., Rashid, A., Filho, F. Reflections on Architectural Connection: Seven Issues on Aspects and ADLs. Workshop on Early Aspects ICSE'06, May 2006, Shanghai, China.
- [13] Garcia, A. et al. Modularizing Design Patterns with Aspects: A Quantitative Study. 4th Intl Conference on Aspect-Oriented Software Development (AOSD'05), Chicago, USA, March 2005.
- [14] Cacho, N., Sant'Anna, C., Garcia, A., Batista, T., Lucena, C. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. 5th Intl. Conference on Aspect-Oriented Software Development (AOSD'06), Bonn, Germany, March 2006.
- [15] Filho, F., Rubira, C., Ferreira, R., Garcia, A. A Quantitative Study on the Aspectization of Exception Handling. In: “Advances in Exception Handling Techniques”, LNCS, Springer, May 2006.
- [16] Rashid, A., Chitchyan, R. Persistence as an Aspect. Proc. of the 2nd Intl. Conf. on Aspect-Oriented Software Development (AOSD'03), USA, March 2003.
- [17] Kulesza, U., Alves, V., Garcia, A., Lucena, C., Borba, P. Improving Extensibility of Object-Oriented Frameworks with AOP, Proc. of ICSR'2006, Springer, LNCS, Torino, Italy, June 2006.
- [18] Krechetov, I., Tekinerdogan, B., Garcia, A., Chavez, C., Kulesza, U. Towards an Integrated Aspect-Oriented Modeling Approach for Software Architecture Design. 8th Workshop on Aspect-Oriented Modelling (AOM.06), AOSD.06, Bonn, Germany.
- [19] Medvidovic, N., Taylor, R. A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Trans. Soft. Eng., 26(1):70-93, Jan 2000.
- [20] Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Pinto, M., Tekinerdogan, B., Clarke, S., Jackson, A. Survey of Analysis and Design Approaches, AOSD-Europe Report D11, 18 May 2005.
- [21] Lago, P., van Vliet, H. Explicit Assumptions enrich Architectural Models. Proceedings 27th ICSE, IEEE, 2005, pp 206-214.
- [22] Tekinerdogan, B. ASAAM: Aspectual Software Architecture Analysis Method. In Proc. of the IEEE/IFIP Conference on Software Architecture, Norway, (2004), pp. 5-14.