

Early Aspects: The Current Landscape

João Araújo
Universidade Nova Lisboa, Portugal

Elisa Baniassad
University of Hong Kong.

Paul Clements
Carnegie Mellon University, USA

Ana Moreira
Universidade Nova Lisboa, Portugal

Awais Rashid
Lancaster University, UK

Bedir Tekinerdoğan
University of Twente, The Netherlands

February 2005

Software Architecture Technology Initiative

Technical Note
CMU/SEI-2005-TN-xxx

Technical Report
Lancaster University
COMP-001-2005

Formatted: English (U.S.)

Unlimited distribution subject to the copyright.

This work is sponsored by the U.S. Department of Defense. The participation of some of the authors is supported by the European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe): EC FP6-2003-IST-2-004349, and by the Portuguese Foundation for Sciences and Technology: POSI/EIA/60189/2004.

The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Copyright 2005 Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (<http://www.sei.cmu.edu/publications/pubweb.html>).

Contents

1	Introduction	1
2	Early Aspects and Domain Analysis	4
2.1	Motivation for Aspect-Oriented Domain Analysis	5
3	Early Aspects and Requirements Engineering	8
3.1	Why AORE?	8
3.2	What is an Aspect at the Requirements-Level?	9
3.3	Aspect-Oriented Requirements Engineering (AORE)	10
3.4	Current Approaches to Early Aspects and Requirements Engineering	12
4	Early Aspects and Architecture Design	17
4.1	Software Architecture and Architectural Views	17
4.2	Architecture-centric Development Activities	19
4.3	Early Aspects and Architecture Design	20
4.4	Current Approaches to Early Aspects and Architecture Design	22
5	Distilling it All: What <i>Is</i> an Early Aspect?	25
5.1	Requirements vs. Concerns	Error! Bookmark not defined.
5.2	Stages of Early Aspects	Error! Bookmark not defined.
5.3	Scattering and Tangling: Cornerstones of Aspect-Orientation	Error! Bookmark not defined.

5.4 **How do we bring all the concepts from these three areas together so we have a conceptually unified treatment?** Error! Bookmark not defined.

5.5 **Vision statement** Error! Bookmark not defined.

6 **Important Research Issues in Early Aspects** 26

6.1 **Basic definition issues** 26

6.2 **Practical application issues** 27

6.3 **In summary** 29

7 **Conclusions and Next Steps** 30

8 **References** 31

{Ed: TOC needs updating}

Acknowledgments

The authors are grateful to all of the participants at the Early Aspects workshops whose contributions, discussion, and energy have led to the contents of this report. We especially thank (in advance) participants at Early Aspects 2005, held in conjunction with AOSD 2005, because they were the first to see this report unveiled and give us specific comments on it.

We also thank John McGregor and Paulo Merson of the Software Engineering Institute, who provided helpful reviews.

The work of some of the authors is supported by the European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe) and by the Portuguese Foundation for Science and Technology.

Abstract

Early aspects are crosscutting concerns that are identified in the early life cycle phases of a software system's development. Nominally these phases include the requirements analysis, domain analysis and architecture design phases. Early aspects cannot be localized and tend to be scattered over multiple early life cycle artifacts or artifact elements (such as sections in a requirements document, or modules in an architectural design). Unless these crosscutting concerns can be localized and managed as distinct concepts, this reduces the modularity of the artifacts in the early life cycle which might consequently lead to serious maintenance problems, or even the failure of the system to meet certain quality attribute goals. This report lays out a conceptual landscape for how the concept of early aspects pertains to domain analysis, requirements engineering, and architectural design. It distills all of the concepts special to each, and provides a conceptually unified view of early aspects in general. It summarizes the current state of the practice, and lays out a research agenda for the future.

1 Introduction

Aspect oriented software development (AOSD) is emerging as an important new approach to software engineering. It sprang forth from a re-thinking of the relationship between modularization (a partitioning of software into discrete, non-overlapping units of implementation) and the long-honored principle of separation of concerns. Any separation-of-concerns criterion will lead to a particular partitioning. A different separation-of-concerns criterion will lead to a different partitioning, as though the software was sliced into pieces in a different direction. Until now, the designer has been forced to choose one, only to watch helplessly as the concerns under the rejected criterion wash across multiple modules. Thus, the system will be well-insulated against changes that spring from the chosen criterion, but the system will be brittle with respect to changes related to the second (rejected) modularization criterion. AOSD solves this problem by introducing the concept of an *aspect* (a concern that cuts across modules) plus programming-level support to (a) implement aspects separately from the modules, isolating them in one place where they may be easily managed and changed, and (b) once defined, to *weave* them back into the modules as though all of the modules each contained a copy of the implementation of the aspect.

A term that elegantly sums up this concept was an early synonym for AOSD: multi-dimensional separation of concerns.

Until now, work in aspects has been fairly limited to the implementation phase of software development, finding concerns that units of implementation have in common and factoring those out as aspects. However, recent work has tried to generalize the concept and apply it to different phases of the life cycle.

In particular, early aspects are crosscutting concerns that are identified in the early life cycle phases of a software system's development. Nominally these phases include the domain analysis, requirements analysis, and architecture design phases. Hence, work in early aspects tends to generalize the definition of "aspect" – no longer is it a concern that cuts across just modules in an implementation, because during these early stages there may not yet be any modules!

Early aspects tend to be scattered over multiple early life cycle artifacts or artifact elements (such as sections in a requirements document, or concerns shared by systems in the same application area in a domain model, or elements in an architectural design). Unless these crosscutting concerns can be localized and managed as distinct concepts, this reduces the modularity of the artifacts in the early life cycle which might consequently lead to serious maintenance problems, or even the failure of the system to meet certain quality attribute goals. Unfortunately, conventional aspect-oriented software development approaches have

almost exclusively focused on identifying the aspects in programming-level artifacts (principally units of implementation) and less attention has been taken on the impact of crosscutting concerns at the early phases of the software development. Obviously, the early software development phases actually set the early design decisions and have a large impact on the whole system. Coping with aspects at the early life cycle phases as such is a primary issue.

And yet, it isn't intuitively clear even what defines an aspect in the absence of implementation artifacts: If an aspect is a cross-cutting concern, what exactly is there that can be cross-cut before architectural elements are defined? Nevertheless, it seems likely that if a reasonable answer can be found to this and similar questions, early aspects can present a valuable tool in the system designer's kit.

In a fashion similar to object-orientation, aspect-orientation is following a cycle of natural progression from programming to other more abstract software development stages. Recent work has shown a significant increase in the interest in aspect-oriented design and the development of guidelines and mechanisms for the purpose. While some initial work exists on applying aspects at the early stages of requirements engineering and architecture design the role of aspects at these stages and their relationship with existing separation of concerns mechanisms such as viewpoints and architectural views is poorly understood. Similarly aspect identification and aspect composition issues at these stages remain largely unexplored. Employing an aspect-oriented approach from the very early development stages can help capture crosscutting properties early on in the software life cycle; hence, improving the quality of the products and reducing adaptation, evolution and maintenance costs. The Aspect Oriented Software Development (AOSD¹) conference itself, the flagship gathering for aspect enthusiasts, has a strong theme dedicated to analysis, maintenance, and processes and methodologies.

Similarly, the flagship gathering currently for early aspect enthusiasts is the Early Aspects (EA) series of workshops. Indeed, the early aspects movement began in 2002 with the organization of the first edition of the "Early Aspects: Requirements Engineering and Architecture Design" workshop, in conjunction with AOSD 2002. Since that time, there has been an edition of EA at every AOSD conference, as well as one at OOPSLA 2004. Early workshops were restricted to participants able to contribute to the group discussions so that an initial EA framework could be delivered to the AOSD community. This strategy helped to focus the group discussions on topics that could contribute to the understanding of the importance of aspect-orientation at this early stage of the software development lifecycle. The aim was to mature the initial ideas before conveying to the broader AOSD, requirements engineering, and software architecture communities the importance of early aspects.

¹ In this report, "AOSD" may refer to the AOSD conference, or abbreviate the more general term aspect-oriented software development. Similarly, "EA" may refer to early aspects as a field of study, or the Early Aspects workshop series. In both cases, usage should be clear by context.

This report is the fruit of that strategy. After four workshops, the current EA organizers (the authors of this report) felt that enough of a coherent picture of early aspects had emerged so that it could and should be captured and distributed for comment and elaboration.

The report is organized as follows. Sections 2, 3, and 4 cover the important concepts of early aspects and how they relate to domain analysis, requirements engineering, and architecture design, respectively. Each of these chapters includes a short compendium of current approaches in use or in development that relate that area to early aspects. Section 5 provides a distillation, our version of a grand unifying theory of early aspects, which is based upon the common concepts found in each of the preceding three chapters. Section 6 lays out a research agenda for the immediate future. Section 7 provides some closing comments and a look ahead.

Comment [clements1]: Comment from John McGregor: Need to define "concern" somewhere, probably early on. Peri Tarr, he says, has a good definition.

2 Early Aspects and Domain Analysis

AOSD has provided an improved and complementary understanding of the separation of concerns principle by explicitly separating and specifying types of concerns, i.e. aspects, that tend to crosscut over multiple components. As a result, an increasing number of software projects explicitly consider the separation of aspects in the application to achieve improved modularity. A close analysis of the software systems shows that very often the same kind of aspects are implemented over and over again. In designing distributed systems, for example, concerns like synchronization, recovery, security, logging, profiling, and monitoring are implemented, and it appears that most of these concerns are crosscutting and would be best implemented as aspects. Despite the fact that many aspects reoccur in many applications no systematic attempt seems to have been undertaken to capture and reuse these aspects. The same (kind of) aspects are either implemented from scratch or at best are reused opportunistically by adapting existing aspect specifications, either informally from the literature or ad-hoc from aspect codes.

Opportunistic reuse might work in a limited way for individual programmers or small groups. However, it does not easily scale up for larger applications and a more *systematic* approach for software reuse is required instead. The generally acknowledged motivation for systematic reuse counts for aspects as well: decreasing development cycle time, increasing quality, and decreasing cost of development.

A logical consideration implies that this need for systematic reuse for aspects will only increase in the future when more and more applications are designed and implemented using aspects. In this context, therefore, it is worthwhile to define an approach for identifying, modeling and reusing aspects that might recur in future applications.

Reusing aspects primarily requires the systematic identification and exploitation of commonality across related aspect specifications. One of the commonly adopted processes for systematic reuse is *domain analysis* which is widely adopted in the software reuse community. *Domain analysis* can be defined as the process of identifying, capturing and organizing domain knowledge about the problem domain with the purpose of making it reusable when creating new systems [3]. The result of domain analysis is a *domain model* which can be reused to implement various applications. Numerous domain analysis methods currently exist, each focusing on increasing the understanding of the domain by capturing the information in reusable model(s) [3][5]. Figure 1 represents the common structure of domain analysis methods as it has been derived from survey studies on domain analysis methods [3][5][6].

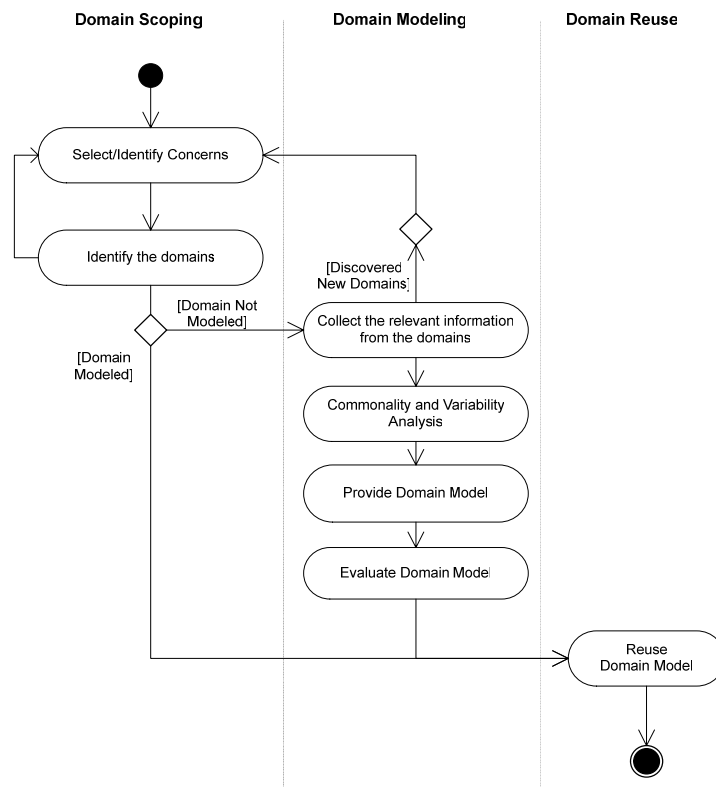


Figure 1. Common structure of domain analysis methods

Conventional domain analysis processes consist generally of the processes *domain scoping* and *domain modeling*. *Domain scoping* identifies the domains of interest, the stakeholders, and their goals, and defines the scope of the domain. *Domain modeling* is the activity for representing the domain, or the domain model. The domain model can be represented in different forms such as object-oriented language, algebraic specifications, rules, conceptual models, etc. Typically a domain model is formed through a commonality and variability analysis to concepts in the domain. A *domain model* is used as a basis for engineering components intended for use in multiple applications within the domain.

2.1 Motivation for Aspect-Oriented Domain Analysis

Although, many successful results have been achieved in these areas, the notion of aspects has not been addressed explicitly yet. We can identify two basic reasons for using domain analysis in the aspect-oriented development:

Identifying aspects from the solution domain: Initially in AOSD, aspects have been basically identified at the code level by code analysis techniques. The general definition of

aspects as being concerns which tend to crosscut over a broader set of modules can, however, be applied throughout the life cycle. As a result, this implies that aspects might occur throughout the whole software life cycle. To avoid the problems related to crosscutting, like reduced maintainability, these crosscutting concerns must be identified and specified separately. To cope with this issue recently more attention has been paid to identifying aspects at the requirements analysis and architecture design. Aspects identified during the requirements analysis, however, might not always be sufficient to cover all aspects. This is because the requirements analysis focuses on the problem domain rather than the solution domain. On the other hand, identifying aspects at the architecture design might be too late. To complete the process of aspect identification we think that it is necessary to also consider aspects in the solution domain. The solution domain has been defined as the domain which includes the solution for the requirements in the problem domain. Unfortunately there is no approach yet for identifying aspects from the domain

Implementing aspects based on solution domain models: Before implementing the aspects it is required that their structure, properties and behavior is sufficiently understood. Otherwise, there will be no solid basis for guaranteeing that the aspect is defined and implemented in a proper way. It appears, however, that in current AOSD practices the analysis of aspects is not explicit and the implementation of aspects depends mostly on the experience and background of the aspect programmer. This might easily lead to unstable aspects which will soon need to be modified. Domain models are generally considered as stable abstractions for implementing concerns. In a similar way, to understand the structure of aspects at the more concrete implementation level we need explicit domain models for aspects.

Obviously, domain analysis has proven its value for a wide range of applications and domains [3]. In our attempts to identify and specify aspects, the first thought that comes to mind is therefore whether we can use existing domain analysis approaches as is. As a matter of fact, domain analysis is a quite generic process for defining any concepts for any given domain. Therefore we could state that it should actually not make a difference whether we are dealing with aspects or not. Aspects, however, are actually a new type of concern that was not taken into account before in existing domain analysis approaches. In addition, this new concern type appears to impose new type of constraints and extensions to identification and specification of domains (for aspects). Further research is needed to investigate whether domain analysis approaches need to be enhanced to cope with this new concern type.

References

- [1] M. Aksit. *How to Find Domains, Aspects and Patterns*, in the proceedings of the 20th International Conference on Software Engineering (ICSE'98), Vol. 2, pp. 20, Kyoto, April 1998.
- [2] M.Aksit & L. Bergmans. *Aspect-Oriented Domain Analysis*, tutorial notes, 2nd Aspect-Oriented Software Development Conference, Boston, USA, March 2003.
- [3] G. Arrango. *Domain Analysis Methods*. In *Software Reusability*, Schäfer, R. Prieto-Díaz, and M. Matsumoto (Eds.), Ellis Horwood, New York, New York, pp. 17-49, 1994.
- [4] G. Booch, J. Rumbaugh & I. Jacobson. *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

- [5] C. Czarnecki & U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.
- [6] Kang, K., Cohen, S., Hess, J., & Nowak, W., & Peterson, S. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1990.
- [7] A. Rashid, A. Moreira & J. Araujo, *Modularisation and Composition of Aspectual Requirements*, in: Proceedings of Second Aspect-Oriented Software Development Conference, pp. 11-21, Boston, US, 2003.
- [8] S. Sutton & I. Rouvellou. *Modeling of Software Concerns in Cosmos*, in: Proceedings of First Aspect-Oriented Software Development Conference, pp. 127-134, Enschede, The Netherlands, 2003.
- [9] B. Tekinerdogan. *ASAAM: Aspectual Software Architecture Analysis Method*, in: Proc. Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, 2nd Aspect-Oriented Software Development Conference, (also submitted for journal publication), March, 2003.
- [10] B. Tekinerdoğan & M. Akşit. *Deriving design aspects from conceptual models*. In: Demeyer, S., & Bosch, J. (eds.), Object-Oriented Technology, ECOOP '98 Workshop Reader, LNCS 1543, Springer-Verlag, pp.410-414, 1999.

3 Early Aspects and Requirements Engineering

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.

As mentioned earlier in this report, AOSD techniques to date have mainly focused on treating crosscutting concerns at the later stages in the solution domain, i.e., design and implementation. There is often a misconception that *design modeling of crosscutting concerns* implies an *early treatment of aspects*. However, to treat aspects during design modelling one has to identify these aspects in the first place, and from a very early stage, so that they are effectively reflected in the requirements specification as well as in the architecture derived from it. Aspect-oriented requirements engineering (AORE) aims at treating broadly scoped, crosscutting properties in a systematic fashion in order to facilitate effective reasoning about their impact in the problem domain as well as modularizing such crosscutting properties in the requirements specification so that they can be effectively mapped, and hence traced, to the solution domain. However, before discussing AORE in more detail, it is important to answer two very important questions:

- Why AORE?
- What is an aspect at the requirements-level?

3.1 Why AORE?

In a fashion similar to architecture, design and programming techniques, requirements engineering approaches have long recognized the need for effective separation of concerns. Viewpoint-oriented techniques [AR1], for instance, partition a system's requirements from various partial, subjective perspectives derived from stakeholders' views. Similarly, use cases [AR2] provide descriptions of the system from the perspective of its use by particular actors in its environment. Goal-oriented approaches [AR3][AR4], on the other hand, drive the elicitation and modularization of a system's requirements from a number of high-level goals which are subsequently refined into more concrete goals. All these approaches, in their own fashion, capture crosscutting views of a system's requirements. It is, therefore, important to ask the question as to why we need aspect-oriented techniques at the requirements level.

Though viewpoints and use cases do provide subjective perspectives on a system, they do not treat non-functional properties, e.g., *security*, *real-time constraints*, *mobility*, etc., in a systematic fashion. These properties often form good candidates for aspects that cut across viewpoints and use cases. Some approaches, e.g., PREView [AR5] do treat non-functional properties explicitly. However, beyond alerting a requirements engineer to the fact that these properties may affect certain viewpoints (and may interact with reference to those), they do not clearly capture how they constrain or influence specific requirements in the system. In other words, they do not specify the compositional relationships between broadly-scoped, non-functional properties and the stakeholder requirements affected by them. Goal-oriented approaches, on the other hand, drive the requirements engineering process from the perspective of systemic goals, which are often non-functional in nature. Consequently, they do not effectively capture the influence of highly functional crosscutting properties, e.g., *information retrieval*, *information update*, etc., on other requirements in the system.

AORE aims to address the above shortcomings by *providing a systematic means for the identification, modularization, representation and composition of crosscutting properties, both functional and non-functional ones, during requirements engineering*. One of the terms to highlight here is *composition*. AORE techniques build upon the strong focus on composability in AOSD by providing a fine-grained specification of how a requirements-level aspect constrains or influences specific requirements in a system. Such a detailed understanding of the composition relationships between aspectual and non-aspectual requirements leads to an improved understanding of their interaction, inter-relationships and conflicts. This, in turn, helps to identify trade-offs early on in the development life cycle and undertake negotiations with the affected stakeholders. Furthermore, the aspectual requirements and their associated trade-offs can be traced to implementation to ensure that they have been preserved in line with the requirements specification that the stakeholders signed off on.

3.2 What is an Aspect at the Requirements-Level?

An aspect at the requirements-level is a broadly-scoped property, represented by a single requirement or a coherent set of requirements (e.g., security requirements), that affects multiple other requirements in the system so that:

- it may constrain the specified behavior of the affected requirements;
- it may influence the affected requirements in order to alter their specified behavior.

For instance, a security requirement may constrain a requirement providing access to certain types of data in the system so that only a certified set of users may access that data. Similarly, another security requirement may influence communication requirements by altering their behaviour to impose encryption constraints.

Note that the requirements affected by a requirements-level aspect may already have been partitioned using abstractions such as viewpoints, use cases and themes [AR6]. Figure ARFig1 shows a requirements-level aspect affecting multiple requirements in such a partitioning.

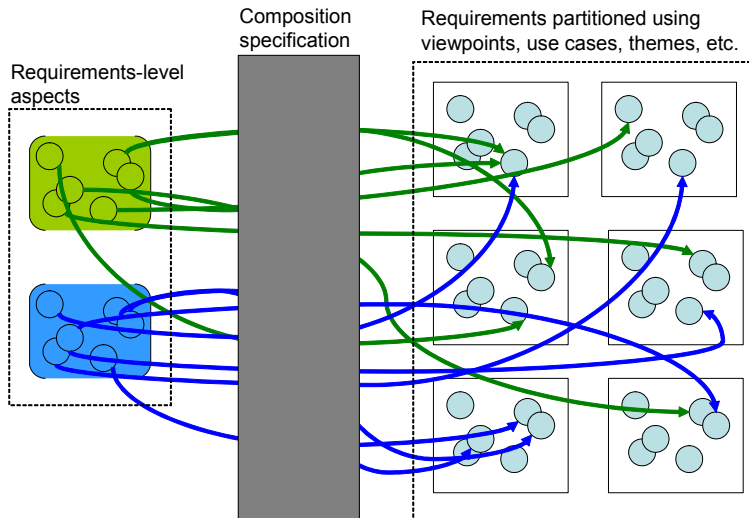


Figure ARFig1: Requirements-level aspects constraining and influencing (via a composition specification) other requirements

3.3 Aspect-Oriented Requirements Engineering (AORE)

As mentioned above, an AORE approach is characterized by four key characteristics:

1. The existence of effective means to identify crosscutting properties in a requirements specification. This may be accomplished by providing intuitive guidelines for the purpose, e.g., [AR7][AR8], identifying specific keywords or action words in a requirements specification, e.g., [AR6] or using specific tool support based on, for example, semantic analysis of requirements documents [AR9].
2. The ability to modularize crosscutting properties pertaining to a particular concern in one requirements-level module, i.e., a requirements-level aspect. Note that such aspects may be based on a multi-dimensional separation [AR10] where the requirements are partitioned symmetrically or they may follow a more classical,

asymmetric, base-aspect separation [AR6][AR7][AR8][AR11]. Nevertheless, it is the ability to modularize such a crosscutting set of requirements that is the distinguishing characteristic and not the symmetry or asymmetry of the modularization mechanism.

3. The provision of suitable means to represent requirements-level aspects. Again, such representation mechanisms may differ, depending on the application domain or availability of relevant tools. As a result, the representation may be graphical [AR6][AR11] or may take a semi-structured format [AR8][AR10].
4. The ability to compose aspectual requirements and non-aspectual requirements to clearly understand the cumulative effect of requirements-level aspects on other system requirements. Note that composition in this sense does not have to yield an *executable requirements specification*. Though this may be desirable in some instances, it is not a defining characteristic of an AORE technique. Instead composition here implies *projecting* the constraints and influences of individual requirements-level aspects on other system requirements based on based on the knowledge inherent in the composition specification (cf. figure ARFig1). The outcome of such a composition is a *synthesis of the various projections* that provides a deeper understanding of the critical needs of the stakeholders, identification of the key properties of a system as well as the various trade-offs affecting the system.

Though not an essential characteristic, an AORE approach should be complemented with effective facilities to trace aspectual requirements and any associated trade-offs (as well as their resolutions) to latter development stages. Often requirements-level trade-offs are resolved by weakening certain requirements in order to strengthen others as a result of negotiations with stakeholders. In such instances, it is essential to specify what a *weakened requirement* or a *strengthened requirement* means so that this understanding can be effectively propagated to the architecture and subsequently to design and implementation [AR12].

3.3.1 A Join Point Model for AORE

Since requirements-level aspects provide additional support for composability, it is important to discuss the notion of a join point model at this abstraction level. Figure ARFig1 shows how an aspect at this level may effect multiple requirements already partitioned into requirements-level modules. However, we can observe that the join points are not something similar to nodes in an object call graph as is the case in aspect-oriented programming (AOP) techniques such as AspectJ [AR13]. Those join points make sense at the programming level. However, at the requirements-level, join points are effectively individual requirements in a viewpoint, a use case or a theme and so on. We can use something similar to wild card matching in AOP techniques by specifying that an aspectual requirements affects not only a particular requirement but also any sub-requirements that refine it [AR8]. Similarly, we may

specify that an aspectual requirement (or a set of them) constrains all the requirements captured by a viewpoint or a use case.

For example, consider a security aspect at the requirements-level which specifies simple login-based authentication for all users of the system. However, this top-level requirement is refined to provide more specific rights for different categories of users, e.g., managers, system administrators, etc. Let us assume that we have a viewpoint-based partitioning of our stakeholder requirements. A composition specification (based on our above definition of a join point model) would take the following form:

Constrain *all requirements specified in all user viewpoints* by the **top-level security requirement**.

Constrain *all requirements in administrator viewpoint* by the **security policy for administrators**.

Constrain *all requirements in manager viewpoint* by the **security policy for managers**.

...

...

If we wish to draw an analogy with join point models for AOP, the italicized elements in the above composition specification are the *pointcuts*, the underlined elements are the *type of advice* and the bold elements are the *advice behavior*. However, note that unlike AOP techniques such as AspectJ (and others), we provide an explicit separation between the requirements-level aspect and its composition specification, i.e., the semantics of the top-level security requirement, security policy for administrators, etc. are specified in a separate requirements-level security aspect and not alongside the composition specification. This provides improved separation of concerns and facilitates localization of impacts resulting from requirements changes and evolution.

In this section, we describe the state of the art of early aspects at requirements and architecture levels. Here we will discuss current techniques and tools proposed by several researchers in order to support the early stages of software development.

3.4 Current Approaches to Early Aspects and Requirements Engineering

3.4.1 Non-Aspect-Oriented Requirements Engineering Approaches

There are several approaches to model requirements, such as use case driven, object-oriented, viewpoint oriented and goal oriented approaches. Goal-oriented [Lamsweerde 2001] and viewpoint-oriented [Finkelstein & Sommerville 1996] approaches address the separation of functional and non-functional concerns.

Well-known methods that incorporate goals are KAOS [Dardenne et al. 1993] and *i** [Yu 1995]. A *goal* is a purpose that the system is supposed to reach. A goal can be expressed informally or formally (e.g., using temporal logic as in KAOS) and include functional and non-functional concerns. On the other hand, PREView [Sommerville & Sawyer 1997], for example, is a viewpoint-oriented requirements engineering method, which helps to separate functional and non-functional properties of a system. A viewpoint in PREView encapsulates partial information about the system obtained from the stakeholders. PREView includes, among others concerns that are non-functional and are specified with a set of requirements.

Nevertheless, these approaches do not deal with crosscutting concerns, or crosscutting requirements, satisfactorily. Requirements are crosscutting by nature, where a requirement may have an effect on several other requirements. Yet, neither do non-aspect-oriented requirements approaches reflect the separation of these crosscutting requirements, nor they present mechanisms to compose such requirements in a successful manner without losing abstraction. The aspect-oriented requirements approaches fill the gap left by the traditional ones. These are described next.

3.4.2 Aspect-Oriented Requirements Engineering Approaches

The Aspect-Oriented Component Requirements Engineering approach (AOCRE) [Grundy 1999] was one of the first to be proposed. Here, there is a categorization of various aspects of a system (e.g., persistence, distribution) that each component makes available to other components or users. It is committed to component based software development, but does not present evidence of its use in other sorts of software development.

The approach in [Dingwall-Smith & Finkelstein 2002] shows the development of a system for runtime monitoring of system goals. KAOS is used to specify the system. They define separated composition rules using Hyper/J, which consequently makes the approach dependent on an implementation. Also, it is domain specific, limiting its applicability.

Cosmos is a schema for the modeling of multidimensional concern-spaces [Sutton & Rouvellou 2002]. It defines any concern as a matter of interest. The schema includes concerns (logical and physical), relationships (categorical, interpretive, mapping, physical), and predicates. Even though the schema favors detailed modeling and modularization, little support is given to composition of concerns.

In [Moreira et al. 2002] and [Brito et al. 2002], separation of crosscutting concerns at the requirements level is achieved by identifying and specifying quality attributes (i.e., higher-level non-functional requirements) and functional requirements (using a use case driven approach). Quality attributes are described using special templates, identifying those that crosscut functional use cases. Extensions to use cases and sequence diagrams are proposed to represent the composition of crosscutting quality attributes with functional requirements.

Composition rules are presented very informally and support for mapping to design and conflict resolution is not considered.

In [Sousa et al. 2004], it is presented an approach that adapts some use-case driven activities of the Unified Software Development Process (requirements, analysis and design workflows). The purpose of this approach is to provide mechanisms that support the separation of crosscutting concerns in artifacts of these workflows. In these activities non-functional concerns are integrated in the use case model.

The Aspect-Oriented Requirements Engineering (AORE) approach [Rashid et al. 2002] proposes a model that supports separation of crosscutting functional and non-functional properties at the requirements level. These crosscutting properties are classified as candidate aspects. It also provides identification of their mapping and influence on later development stages. The approach is refined in [Rashid et al. 2003] with PREView and XML, where detailed composition rules for aspectual requirements are defined and also separated. The XML composition rules use a list of constraint actions and operators, with well-defined semantics. They are used to specify how an aspectual requirement influences or constrains the behaviour of a set of non-aspectual requirements. Also, early separation of crosscutting requirements makes it possible the determination of their mapping and influence on artifacts at later development stages. Besides, an elaborated conflict resolution scheme is presented. The approach is supported by a tool called ARCaDe.

Additional support for traceability in the AORE model is provided by the PROBE framework [Katz & Rashid 2004], which generates proof obligations based on temporal logic that should hold for an aspect-oriented system from the initial aspectual requirements and their related trade-offs. The temporal logic assertions can be used as input to formal methods tools such as model-checkers or as the basis for deriving test cases.

Another approach that combines viewpoints and aspects is found in [Araújo & Coutinho 2003]. Here, a viewpoint-oriented requirements approach with UML is extended to include aspect-oriented concepts. This is observed during the definition of both non-functional requirements and use cases. NFRs that crosscut a number of viewpoints or use cases are non-functional candidate aspects. Use cases that are included by or extend more than one use case are called aspectual use cases. Also, uses cases that crosscut several viewpoints are also called aspectual use cases.

[Whittle et al., 2003] and [Whittle & Araújo 2004] present an aspect-oriented scenario modelling approach, to be used at the requirements level. Aspectual scenarios are modelled using Interaction Pattern Specifications [France et al 2004] and composed with non-aspectual requirements. The process model presented also included the generation of state machines from composed scenarios (represented by sequence diagrams), to be used to create prototypes for stakeholder's requirements validation. The generation of state machines from sequence diagrams is realised using the Synthesis approach, described in [Whittle & Schumann 2000].

In [Araújo et al. 2004], a variation of this approach is proposed, where the composition is realised at state machine level. Advantages and disadvantages of the two approaches are still to be pondered.

[Baniassad & Clarke 2004b] propose Theme to provide support for aspect-oriented analysis through Theme/Doc. Analysis is carried out by first identifying a set of actions in the requirements documentation that are, in turn, used to identify crosscutting behaviors. The authors define actions as sensible verbs for the domain. An action is a potential theme, which is a collection of structures and behaviors that represent one feature. A tool is provided to create graphs of the relationships between concerns and the requirements that mentioned those concerns. One problem with Theme/Doc is that it does not provides enough support for requirements of large scale systems. Nevertheless, in [Baniassad & Clarke 2004a] initial research is discussed about using several latent clues in requirements documentation to help scale document concern-views.

The approach described in [Brito & Moreira 2003] presents a model to handle advanced separation of concerns during requirements engineering. The novelty introduced here is, on the one hand, the use of existing catalogues to help identify and specify concerns and, on the other hand, a new composition process with the introduction of match points, dominant aspects and composition rules using LOTOS operators. [Brito & Moreira 2004] extends the approach by, firstly, using the NFR framework catalogue, and, secondly, exploring a refinement of the LOTOS composition rules considering this framework. Notice that this approach treats all concerns (crosscutting or non-crosscutting) in a uniform manner.

[Yu et al. 2004] propose an approach for discovering aspects from relationships between functional and non-functional goals. This is realized by means of a goal model, where functional and non-functional requirements are represented through goals and softgoals, plus tasks that contribute to their satisfaction. The resulting graph of the model is analyzed to identify aspects.

References

- [AR1] A. Finkelstein and I. Sommerville, "The Viewpoints FAQ." *BCS/IEE Software Engineering Journal*, 11(1), 1996.
- [AR2] I. Jacobson, *Object-Oriented Software Engineering - a Use Case Driven Approach*: Addison-Wesley, 1992.
- [AR3] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*: Kluwer, 2000.
- [AR4] A. Dardenne, A. Lamsweerde, and S. Fickas, "Goal-directed Requirements Acquisition", *Science of Computer Programming*, Vol. 20, pp. 3-50, 1993.
- [AR5] I. Sommerville and P. Sawyer, *Requirements Engineering - A Good Practice Guide*: John Wiley and Sons, 1997.
- [AR6] E. Baniassad and S. Clarke, "Theme: An approach for aspect-oriented analysis and design". In 26th Int'l Conf. Software Engineering (ICSE), (Edinburgh, Scotland), 2004.

- [AR7] A. Rashid, P. Sawyer, A. Moreira, and J. Araujo, "Early Aspects: A Model for Aspect-Oriented Requirements Engineering", Proc. RE, 2002, IEEE Computer Society Press, pp. 199-202.
- [AR8] A. Rashid, A. Moreira, and J. Araújo, "Modularisation and Composition of Aspectual Requirements", Proc. AOSD, 2003, ACM, pp. 11-20.
- [AR9] A. Sampaio, N. Loughran, A. Rashid, P. Rayson, "Mining Aspects in Requirements", Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, AOSD 2005, Chicago, IL, USA.
- [AR10] A. Moreira, J. Araújo, A. Rashid, "A Concern-Oriented Requirements Engineering Model", Proc. CAiSE 2005, To Appear.
- [AR11] I. Jacobson, P.-W. Ng, "Aspect-Oriented Software Development with Use Cases", Addison Wesley 2005.
- [AR12] S. Katz, A. Rashid, "From Aspectual Requirements to Proof Obligations for Aspect-Oriented Systems", Proc. RE, 2004, IEEE CS Press, pp. 43-52.
- [AR13] AspectJ Project, <http://www.eclipse.org/aspectj>, 2005.

4 Early Aspects and Architecture Design

As we saw in Section 2, early aspects and domain analysis is about using the techniques of domain analysis to identify and capture aspects that often are useful in systems that belong to the domain(s) being analyzed. That is, aspects in domain analysis correspond to aspects in implementation, just identified much earlier.

Section 3 showed us that the relationship between early aspects and requirements is that aspects can be used to identify concerns that cut across different requirements. A requirements specification is the end result of a de facto partitioning of requirements information into (at the coarse level) sections of the document and (at the fine level) individual requirements. Requirements specifications for large complex systems are (just like the systems themselves) sometimes time-consuming to change. Here, aspects can be used to factor out concerns that cut across requirements, and be used to make changing the specification easier. An aspect identified in this way may or may not carry forward into the implementation – that is, although the requirement(s) expressed by the aspect will be implemented, there is no obligation that it be implemented as an aspect (unless the requirements specification states otherwise). Its primary use is to make specifying the requirements more robust.

In software architecture, aspects can play both roles. First, some background. The next section introduces software architecture and perhaps its most important concept: views.

4.1 Software Architecture and Architectural Views

A software architecture is the structure or structures of a software system; these structures comprise elements, the externally-visible properties of those elements, and the relationships among them [Bass 03]. Modern treatments of software architecture embrace the concept of *views*. A view is a representation of a set of system elements and the relations associated with them [Clements 02]. Thus, a view restricts our attention to a particular kind of element (such as layers, or clients and servers, or communicating processes) and the corresponding relation types (allowed to use, request/reply, synchronization and communication). Views essentially are windows into the architecture, each of which is intellectually manageable and together providing a holistic picture.

Views provide the architect with an engineering handle to design the system in a way to meet its behavioral and quality attribute goals. Views also provide a way to represent the architecture so that its stakeholders can understand it. For example, if an architect is interested in making sure that the system can be built in incremental subsets, he or she will

take care to restrict usage among software elements. If everything is allowed to use everything else, then every increment of the system must include every software element. And so a design amenable to incremental development will have a carefully-crafted uses view that shows how each software element may only use a small number of other elements, explicitly identified. After the architect has engineered the uses view, it can serve as part of the architecture documentation so that (for example) developers of an element know what other elements they are allowed to use, and the testers will know what elements are going to be present in each increment.

That is but one example. To assure run-time performance usually requires the architect to explicitly engineer a process or thread or deployment view. To assure modifiability, the architect will have to construct a module decomposition view that purposefully assigns responsibilities to discrete modules. And so forth.

There are an unlimited number of architectural views possible, but in practice an architect will choose a handful (six or less) with which to design the system and communicate that design to others. Some approaches prescribe a fixed set of views for the architect to work with (such as Kruchten's 4+1 view approach [reference] that has become the basis of the Rational Unified Process) but the current trend is to give the architect freedom to choose the views most helpful to the design task at hand and the stakeholders who are the architecture's consumers.

[Insert IEEE-1471 conceptual framework picture here, and explain it briefly.]

Each architectural view reflects a partitioning of the system into architectural elements. A communicating-process view is a picture that prescribes partitioning all of the run-time behavior of the system into a set of processes. A layers view prescribes partitioning the set of software into discrete layers. A client-server view prescribes a partitioning of the run-time functionality into clients and servers. And so forth.

It is almost always the case that the elements appearing in one view will have a strong correspondence to the elements in another view. For example, a very useful view is a deployment view that shows how software elements are allocated to hardware resources – what software runs on what hardware. The software elements shown in this view can usually be mapped straightforwardly to software elements in one of the so-called “component and connector” class of views that show the system as a set of run-time units of functionality, interacting and communicating with each other. Or, perhaps the layers in a layered view correspond to tiers in a multi-tier view (which usually includes information about allocation to hardware). Another useful view is a module decomposition view that hierarchically describes all of the units (and sub-units and sub-sub-units, etc.) of implementation that have to be built, with an assignment of functional responsibility to each. One of these modules might be a client whose many instantiations would be captured in a client-server view. The relation between these two views, then, is that one module maps to (is the implementation of)

many clients. And so forth. One of the documentation obligations of the architect is to explain the correspondence between elements in different views.

4.2 Architecture-centric Development Activities

When software architecture plays a central role in the design of a system, it brings with it a number of related activities. Briefly, these are:

- **creating the business case for the system:** the architect uses the business case to identify stakeholders and the major driving goals behind the system's construction. These goals will eventually be refined into quality attribute goals (such as for performance or security) that will drive the design.
- **understanding the architecturally-significant requirements:** from the requirements engineering task, the architect needs to know which requirements are the most important and which will have the largest impact on the architecture. One way to think about this is to imagine a requirement and imagine the number of ways an architect might design the system to satisfy that requirement. If there are many known ways to achieve it, it is not what we would call an architectural driver. But if the requirement brings with it few known ways to achieve it, then this is a requirement that will shape the architecture.
- **creating and/or selecting the architecture:** this, of course, is the heart of architectural design. Currently, architectural patterns (e.g., [Buschmann]) form the basis for pre-packaged architectural design decisions. Architectural tactics represent a finer-grained approach to imbuing the architecture with quality attributes, and the Attribute-Driven Design (ADD) method [ref] employs these to the fullest.
- **documenting and communicating the architecture:** although the Unified Modeling Language (UML) is what most people think of when they think of documenting an architecture, UML is only a notation and does not prescribe what to write down. As we discussed earlier, modern approaches to documentation are centered around views, but information beyond views is also necessary – for example, the architect must record how the elements in different views are related to each other. ANSI/IEEE-1471-2000, the recommended best practice for documenting architectures of software-intensive systems [ref] exemplifies this philosophy. The SEI's "views-and-beyond" approach [Clements 02] goes a step further to suggest actual documentation contents and layout.
- **analyzing or evaluating the architecture:** one of the primary reasons that software architecture has emerged as an area of conceptual importance is that it can be evaluated to see if the resulting implementation will deliver the desired behavioral and quality attributes. Evaluating an architecture can find design problems before

they become embedded in the system and are orders of magnitude more difficult and expensive to change. A best-of-breed technique is the Architecture Tradeoff Analysis Method (ATAM) [ref], which gathers an architecture's stakeholders for a (roughly) 3-day exercise in eliciting scenarios that express the desired qualities of the architecture, prioritizing the scenarios, and then analyzing the important ones against the architectural decisions in place.

- **implementing the system based on the architecture and ensuring that the implementation conforms to the architecture:** these are downstream tasks to ensure that the architectural vision is carried out.

Of course, it is possible (and desirable) to combine steps. For example, the Twin Peaks model [Nuseibeh 2001] focuses on developing requirements and architectures in parallel to develop an understanding of the system's technical likelihood much earlier, discover additional requirements and constraints and evaluate different design solutions.

4.3 Early Aspects and Architecture Design

As we asserted above, we can use aspects in the architectural design both to aid in capturing the architecture, and to look ahead to implementation-time aspects.

How can aspects help us represent an architecture? We can use aspects to factor out information or properties common about elements that appear (or that have corresponding elements that appear) in more than one view. Alexander Ran writes eloquently about architectural texture [ref], which might be roughly described as a set of fine-grained design decisions that apply to a whole set of elements – for example, elements carrying out transactions in a shared database are all required to follow certain protocols in order to achieve rollback in the case of failure. This shared protocol could be captured in an aspect. We would wish to describe these texture decisions in one place, and not for every element to which they apply. Thus, just like aspects in requirements, we can make our architectural specification more robust and easy to change.

How can aspects in architecture look ahead to aspects in implementation? Architecture is largely about the achievement of quality attributes such as performance, reliability, maintainability, the ability to be developed in parallel by separate teams, security, and so forth. If simple functionality (i.e., calculating the correct answer) were our only goal, a single monolithic block of code would do the job. It is only when we add these other goals that partitioning into elements and careful orchestration of the relationships and interactions among them becomes important. Current approaches to architecture concentrate on employing architectural patterns [Buschmann] to apply pre-packaged design solutions to recognizable design situations. Tactics [ref] provide a finer-grained approach to achieve quality attributes. Whereas patterns can be thought of as a pre-packaged coherent set of design decisions, tactics correspond more to general approaches. For example, replication is

a tactic to achieve reliability, and will probably be used in most patterns for warm- or hot-failover. Separation of concerns is a tactic to achieve modifiability, which appears in many patterns concerned with making changes easier. Tactics can be applied in concert with each other when the goal is a combination of quality attributes, as is almost always the case.

Once an architect has identified the quality attribute goals that the architecture must be structured to achieve, tactics can be chosen to achieve them. Now the architecture is imbued with (a) partitioning decisions reflected in the various views, and (b) tactics that will apply to one or more elements (or groups of interacting elements) in each view. And voila – the tactics that apply to more than one element are by definition a concern that cuts across those elements. That recognition can lead straightforwardly to identification of aspects that can be factored out and isolated so that the system is easily changed should the architect wish to change the choice of tactics sometime in the future.

To generalize, design decisions that apply to more than one element (such as “texture” decisions) might well be factored out into their own architectural element to set policy for a set of elements.

In practice, partitioning and choice of patterns or tactics work hand in hand. Often a tactic or pattern will in fact prescribe a partitioning; a classic example is the Model-View-Controller pattern [ref] that prescribes a division of functionality into three elements with separate concerns. Other times a tactic can be applied to the set of elements that the architect has created through an a priori partitioning; for example, applying replication to a set of service-providing elements creates a standby copy of each ready to take over in case the primary fails. In either case, the architect can factor out the tactics that apply to more than one element, and prescribe to the downstream implementers that those choices be implemented or enforce using aspect-oriented programming.

How can aspects play a role in architecture-centric development? Imagine the set of architecture-centric development practices (listed in the previous section) imbued with the full capabilities of early aspects.

- **creating the business case for the system:** the architect can analyze this to see if there are any overriding concerns in the business case that would otherwise cut across the elements he or she has in mind.
- **understanding the architecturally-significant requirements:** the requirements engineering task delivers to the architect a set of identified aspects. Some of these aspects describe concerns that cut across requirements, while others actually prescribe aspects that should be designed into the system.

- **creating and/or selecting the architecture:** here, the architect uses the aspects from the requirements step as a starting point, and adds additional aspects to the architectural mix to achieve desired flexibility.
- **documenting and communicating the architecture:** as described earlier, aspects are used to factor out commonality in the descriptions, but design aspects are documented in their own right, very possibly in an aspect view. In addition to showing and defining the aspects themselves, the architect must also show which elements are affected by the aspects – that is, the join points. It might be possible to show this relationship in the aspect view itself, should there be one, or in a separate piece of documentation that establishes a correspondence between aspects and elements to which they apply.
- **analyzing or evaluating the architecture:** here, the evaluation is augmented to include the architecture's (a) adherence to required aspects; (b) prescriptive use of aspects in the implementation in order to achieve specific quality goals.
- **implementing the system based on the architecture and ensuring that the implementation conforms to the architecture:** the implementers now receive a set of aspects that they are required to implement, in addition to the ones that they may discover in the normal course of their implementing duties.

The figure below illustrates. (Figure TBD. Shows a flow through the activities, showing how aspects are created, absorbed, or carried through at each point.)

4.4 Current Approaches to Early Aspects and Architecture Design

Software architectures include the early design decisions and embody the overall structure that impacts the quality of the whole system. The conventional approaches have mainly focused on separating the concerns that fit agreeably into architectural components. The approaches described below try to tackle the problem of modeling these crosscutting concerns at architectural level.

In [Tekinerdogan 2003], it is proposed the Aspectual Software Architecture Analysis Method (ASAAM) with the aim to explicitly identify and specify these architectural aspects and make these clear early in the software development. ASAAM extends Software Architecture Analysis Method (SAAM) [Kazman et al. 1996]. ASAAM differs from SAAM as it introduces a set of heuristic rules that help to derive architectural aspects and the related tangled architectural components from scenarios. Scenarios are classified into direct scenarios, indirect scenarios, aspectual scenarios, and architectural aspects. Also it is

provided a characterization of components into cohesive component, composite component, tangled component, and ill-defined components.

In [Cooper et al. 2003], it is presented an overview of the Formal Design Analysis Framework (FDAF), that is intended to support the design and analysis of non-functional properties using a mixture of existing semi-formal and formal methods. The aims of the framework include supporting the designer to identify, analyze, and negotiate conflicting properties using an extended version of the NFR Framework [Yu 1995], define a formal model for specific aspects of the system (such as security and adaptability), and analyze formal models. The FDAF integrates current research in aspect-oriented design and uses the concepts of the NFR Framework to support the identification, analysis, and negotiation of incompatible non-functional properties. A variety of notations have already been used to describe architectures including UML and formal methods to support the rigorous analysis of a design.

In [Kulesza et al 2004], it is depicted the development process of an aspect-oriented generative approach. It involves the specification of a generic AO architecture, in the context of domain engineering. The process is organized into domain analysis; domain design; and domain implementation, as in [Czarnecki and Eisenecker 2000]. The Domain Analysis phase consists of the definition of the domain, the identification and modeling of common and variable features of the domain concepts, and the identification and modeling of crosscutting features of the domain. The Domain Design phase includes the specification of the generic AO architecture, the specification of each non-aspectual and aspectual components of the AO architecture, the identification and specification of domain specific languages (DSLs), and the specification of the configuration knowledge. The Domain Implementation phase comprehends the implementation of the DSLs, the implementation of the AO architecture and components, and the implementation of a code generator.

In [Bass et al. 2004], a method is presented to derive software architecture from its quality attribute requirements (represented as quality attribute scenarios) by using quality attribute models called architectural tactics. The method begins with quality requirements and moves to a design. During this derivation process, architectural aspects can be discovered. These architectural aspects are candidate aspects to be carried through design. Architectural aspects are architectural views consisting of architectural pointcuts and architectural advice. The method provides the architect with guidance as to which architectural tactics should be used to solve the problem.

In [Pantoquilha & Moreira 2004], it is proposed a layered approach for the specification of aspects and their integration at the logical architecture design level, based on multiple views and crosscutting aspectual views composed by architectural aspects. To demonstrate the concept applicability, they apply the ideas to a data warehouse architecture design where the metadata is represented as an aspectual, crosscutting view.

5 Distilling it All: What *Is* an Early Aspect?

This section is not yet written. It is hoped that discussions at Early Aspects 2005 will help to fill it in. The intent is that we find a common ground among the definitions and themes found in each of the preceding sections, and capture that unified, generalized treatment of early aspects in this section.

6 Important Research Issues in Early Aspects

The first papers addressing crosscutting properties at the requirements and architecture levels date from 2002 (<list of refs> [RE 2002], [SEKE 2002], etc). The initial work developed on aspect-oriented requirements engineering (AORE) took non-functional requirements as a starting point [refs]. By their very nature, quality attributes affect several components of a system. Therefore, it is not surprising that research has been conducted to investigate the relationship between quality attributes and aspects. Investigation into aspects and architecture has followed the same thread, since a fundamental purpose of architecture is to endow a system with quality attributes that extend beyond its mere ability to produce the correct result.

The work done so far has helped to identify a number of key research topics that still need to be addressed. These can be grouped loosely into a set of issues that have to do with defining the very nature of EA, while others deal with issues relating EA to other fields or with the practical application of EA principles. They are discussed, in turn, below.

6.1 Basic definition issues

Ontology. An important issue is to develop a common and consistent ontology that will include concepts from which we can assemble a theory for this particular domain (or possibly for aspect-orientation in general) and for enabling knowledge sharing and reuse [Gruber 1993]. It is not clear if the best option is to create a new terminology for early aspects, or to accept the relatively standard terminology in aspect-oriented programming, mainly influenced by AspectJ. Therefore, a basic question to answer is if EA should propose a programming-independent terminology. Traditionally, the concepts for a new paradigm appear first related to implementation level. From the past we have two different examples that may help in the decision: while structured programming terminology has not been used directly in structured analysis and design, the object-oriented programming concepts were basically transferred to the various activities of the software development lifecycle. Advantages and disadvantages for both are well-known. So, before an ontology can be put forward, this question needs attention.

Semantics. If an ontology is to be widely accepted, the concepts it describes should have an unambiguous meaning, needing therefore a precise semantics. At this early stage, it will be difficult to define formal semantics for requirements (as many authors have shown during the early 90's [refs]), but it is possible to use a rigorous systematic approach to specify concerns (crosscutting or non-crosscutting) for requirements and architecture level compositions,

transformations and mappings. Having defined an ontology with precise semantics for its concepts, developers and tool constructors have the ideal ground to contribute to make EA commercially accepted.

Identification of aspects. The identification of aspects is another key area of research. In particular, how are early aspects identified and what is a good early aspect? How do existing requirements engineering techniques contribute to help answering these questions? The need to clarify the relationship between early aspects research and research in requirements engineering and architecture design is of major importance. Some of these techniques, for example, goal-oriented approaches [Lamsweerde] and architectural styles [Buschmann], also attempt to deal with the problem of crosscutting concerns. Also, work on business and organizational modelling [Yu], on ethnography [Suchman 1987] and organisational semiotics and its application to requirements elicitation [Lui 2000, Stamper 1994] might be useful to investigate. It is important to understand what lessons early aspects techniques can learn from these approaches. Equally important may be the contribution that Early Aspects can give to more stable RE approaches.

Mappings. Another important issue for Early Aspects is to define the mappings between requirements and architecture aspects. In particular, how are aspects at the requirements analysis level mapped into architectural aspects, and how do they influence the architecture that can be derived from an aspect-oriented requirements specification?

6.2 Practical application issues

Refinement. When defining mappings that express how a more abstract model can be transformed into a more detailed representation, it is important to guarantee that the latter model is conformant with the former. This relationship is known as refinement. A mapping accompanied by a document that describes the rationale of the choices made, are the essential instruments to verify if a refined model is conformant with its abstraction. Several types of refinement exist and should be investigated in the context of early aspects. Refinement and conformance form the basis of traceability, documenting the reason why a given design was done in a certain way.

Traceability. It is important to be able to understand how a business goal relates to each system requirement, or how a requirement relates to a design artefact, or how a design artefact relates to lines of code. The ability to relate elements belonging to different abstraction representations is known as traceability. Aspects at the requirements level may not necessarily map onto aspects at the design and implementation level. They might map onto a system feature/function (e.g. a simple method, object or component), onto a decision (e.g. a decision for architecture choice) and design, or onto a design (and therefore implementation) aspect. (For a concrete example please [aosd 2003].) Furthermore, new solution domain aspects might arise during these later development stages. It is therefore important to understand how early aspects are handled in later stages of the development and how

implementation aspects are traced back to problem domain requirements. In order to support traceability from requirements specifications throughout implementation artefacts, and vice-versa, a traceability reference model will need to be developed to ensure continued alignment between stakeholder requirements and the various outputs of the development process.

Refactoring. As part of the refinement process a series of refactorings may be created. A refactoring is a behaviour preserving transformation that shows how a more detailed representation conforms to the highest-level representation abstraction. Java legacy code, for example, can benefit from the advantages brought by aspect-oriented programming. Therefore, it seems natural to refactor existing legacy representations into aspect-oriented representations, in order to support improved evolution, reuse and modularization.

Validation. A need of every engineer is to know what properties the artefacts s/he is creating is satisfying. Or, in other words, validation is concerned with showing that requirements actually define the system that the customer wants. Additionally, the aim should also be to verify that the resultant system satisfies and preserves the properties specified by early aspects. But, how can we validate the outputs produced at this level of abstraction? When executable models are obtained, then prototyping or simulation tools can be used. However, executable models are difficult to produce at this early stage of development. When precise refinements are defined, then we may have this task facilitated. An interesting path is to explore how the MDA (Model Driven Architecture) [MDA] framework can be used to define a set of transformations that progressively refine more abstract models into platform specific models that are easier to test and validate.

Conflict management. Conflicting crosscutting concerns, i.e. concerns that contribute negatively to each other, may have to be composed together with respect to the same base module or subsystem. When this happens, a balance between those concerns must be reached, as we will not be able to maximize them. For example, security and response time contribute negatively to each other. This means that we cannot expect the best performance if a very sophisticated security technique is applied, as it would certainly slow down the response time. It is important to identify these conflicting situations as early as possible to facilitate negotiation and decision-making among stakeholders, since these situations might lead to a revision of the requirements specification (stakeholders' requirements, aspectual requirements or composition rules). If this happens, then the requirements must be recomposed and any further conflicts arising must be resolved. Their identification during implementation may become more expensive, since it is during the early phases that a more direct contact is maintained with the stakeholders. Therefore techniques and tools that help us managing this problem are most welcome. Several alternate ways to handle conflicting situations have been proposed [AOSD 2003, IB-EA-2003], but no effective solution has yet been developed.

Multidimensionality. Another challenge is to create a multidimensional approach for early aspects. Existing aspect-oriented requirements engineering approaches are largely two-

dimensional, where functional requirements serve as the base decomposition with non-functional requirements cutting across them. The architecture choices are then mainly guided by the non-functional requirements, so that the system quality attributes can be satisfied. These asymmetric approaches suffer from the tyranny of dominant decomposition. Thus, producing a symmetric RE approach that eliminates the dominant decomposition through uniform treatment of the various types of concerns in a system is a challenge that needs to be overcome.

Integration of tools and techniques. <this has been discussed a little in several other points, such as identification, validation. Not sure we need it explicitly here>

6.3 In summary

Existing aspect-oriented approaches [refs] do not effectively address many of the above issues. These approaches are too young and are still being validated to demonstrate that they can be successfully applied to real case studies. Compared with more traditional RE approaches, they may be seen as too naïve and more elaborated models with well-defined semantics still need to be developed.

The above, non-exhaustive, list of outstanding research issues, shows that a whole area of research is open to a number of approaches addressing these issues. Whatever the outcome of such further research, it is already clear that EA tackle a very important problem at the requirements engineering and architecture design stages and can provide improved support for requirements engineers and architects to reason about their specifications and architectures.

7 Conclusions and Next Steps

It is the hope of the authors that this report will constitute a living document, one that can and will be updated as more knowledge is gathered. about the nature and application of early aspects.

The EA strategy so far has consolidated the initial ideas into a sufficiently robust core set of principles to permit an opening of the field to more general research. The strategy now is to invite a broader field of researchers to participate in the refinement of the topic and a faster dissemination of EA. The process of resolving these issues will, in turn, contribute to the enlargement of the EA community.

Comment [clements2]: These two paragraphs are Ana's, slightly modified from her Section 7.

As a corollary of this opening, the EA workshops themselves can henceforth be organized as open workshops as a means of attracting more participants, making Early Aspects better known and more widely used, and prosecuting the research agenda outlined in the previous section.

The full potential of EA is only now beginning to be glimpsed, but early indications are that it could become a critically valuable resource for software system designers. Meanwhile, the Early Aspects workshops continue to be the primary gathering place for researchers and practitioners interested in this novel concept. As of this writing, AOSD 2005 will host the fifth workshop, and proposals are being submitted to the 2005 Requirements Engineering conference, OOPSLA 2005, and the 2005 Working IFIP/IEEE Conference on Software Architecture (WICSA).

8 References

URLs are valid as of the publication date of this document.

T. R. Gruber. "A translation approach to portable ontologies", Knowledge Acquisition, 5(2):199-220, 1993

A. Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour", 5th International Symposium on Requirements Engineering, 2001, IEEE Computer Society Press, pp. 249-261.

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*: John Wiley & Sons, 1996.

L. Suchman, Plans and situated action: the problem of human-machine communication. Cambridge: Cambridge University Press, 1987

K. Lui: Semiotics in Information Systems Engineering. Cambridge University Press, 2000

N. Stamper: Social Norms in Requirements Analysis - an outline of MEASUR. in Jirotko, M. and Goguen, J. (eds.): Requirements Engineering: Technical and Social Aspects. Academic Press (1994)

E. Yu and J. Mylopoulos: "Using Goals, Rules, and Methods To Support Reasoning In Business Process Reengineering", Int. Journal of Intelligent Systems in Accounting, Finance and Management, special issue on Artificial Intelligence in Business Process Reengineering, 5(1):1-13, January 1996, John Wiley & Sons

[MDA, 2003] MDA (2003). MDA Guide (Draft Version 2). Object Management

Group. OMG document ab/2003-01-03.

A. Moreira, J. Araújo, and I. Brito, "Crosscutting Quality Attributes for Requirements Engineering", Proc. SEKE, 2002, ACM Press, pp. 167-174.

William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992.

Blair, G., L. Blair, A. Rashid, A. Moreira, J. Araujo and R. Chitchyan. Engineering Aspect-Oriented Systems. Chapter in book on Aspect-Oriented Software Development. Addison-

Wesley, ISBN 0-321-21976-7, pp379-406, Editor(s): M. Aksit, S. Clarke, T. Elrad, R. Filman. (2004)

M. Pantoquilha, A. Moreira. Aspect-Oriented Logical Architecture Design - A Layered Perspective Applied to Data Warehousing. Desarrollo de Software Orientado a Aspectos (DSOA'2004), Sponsored by AOSD-Europe, in conjunction with JISBD 2004, Malaga, 2004

A. Moreira, J. Araújo. Handling Unanticipated Requirements Change with Aspects. Software Engineering and Knowledge Engineering Conference (SEKE'04), Banff, Canada, June 2004.

I. Brito, A. Moreira. Integrating the NFR framework in a RE model. Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design, workshop of the 3rd International Conference on Aspect-Oriented Software Development, Lancaster, UK, 22-26 March 2004.

(From Previous Chapter 6)

References

[Araújo & Coutinho 2003] J. Araújo and P. Coutinho, "Identifying Aspectual Use Cases Using a Viewpoint-Oriented Requirements Method". Early Aspects 2003: Aspect-Oriented Requirements Engineering and Architecture Design, Workshop of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), Boston, USA, March 2003.

[Araújo et al. 2004] J. Araújo, J. Whittle and D. Kim, "Modeling and Composing Scenario-Based Requirements with Aspects", Proceedings of the 12th IEEE International Requirements Engineering Conference (RE), Kyoto, Japan, IEEE CS Press, September 2004.

[Baniassad & Clarke 2004a] E. Baniassad, and S. Clarke, "Investigating the Use of Clues for Scaling Document-Level Concern Graphs", Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design, workshop of the OOPSLA 2004, Vancouver, Canada, October 2004.

[Baniassad & Clarke 2004b] E. Baniassad, and S. Clarke, "Theme: An approach for aspect-oriented analysis and design", 26th International Conference on Software Engineering (ICSE), IEEE Press, Edinburgh, Scotland, May 2004.

[Bass et al. 2004] L. Bass, M. Klein, L. Northrop Identifying Aspects using Architectural Reasoning, Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design, workshop of the 3rd International Conference on Aspect-Oriented Software Development, Lancaster, UK, 22-26 March 2004.

[Brito & Moreira 2003] I. Brito, A. Moreira. Advanced Separation of Concerns for Requirements Engineering. VIII Jornadas de Ingeniería de Software y Bases de Datos (JISBD), Alicante, Spain, 12-14 November 2003.

- [Brito & Moreira 2004] I. Brito, A. Moreira. Integrating the NFR framework in a RE model. Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design, workshop of the 3rd International Conference on Aspect-Oriented Software Development, Lancaster, UK, 22-26 March 2004.
- [Brito et al 2002] I. Brito, A. Moreira, J. Araújo, "A Requirements Model for Quality Attributes", Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, Workshop of the 1st International Conference on Aspect-Oriented Software Development, University of Twente, Enschede, Holland, 22-26 April, 2002.
- [Chung et al. 2000] L. Chung, B. Nixon, E. Yu and J. Mylopoulos, "Non-Functional Requirements in Software Engineering". Kluwer Academic Publishers, 2000.
- [Clarke et al. 2000] E.M. Clarke, O. Grumberg and D.A. Peled, "Model Checking", MIT Press, 2000.
- [Clarke & Walker 2001] S. Clarke and R. J. Walker, "Composition Patterns: An Approach to Designing Reusable Aspects". Proceedings of the 23rd International Conference on Software Engineering (ICSE), Toronto, Canada, IEEE CS Press, pp. 5-14, May 2001.
- [Clarke & Walker 2002] S. Clarke and R.J. Walker, "Towards a standard design language for AOSD". Proceedings of the 1st International Conference on Aspect Oriented Software Development (AOSD), ACM Press, pp. 113-119, April 2002.
- [Cooper et al. 2003] K. Cooper, L. Dai, Y. Deng, J. Dong, Towards an aspect-oriented architectural framework, Early Aspects 2003: Aspect-Oriented Requirements Engineering and Architecture Design, Workshop of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), Boston, USA, March 2003.
- [Czarnecki, U. Eisenecker 2000] K. Czarnecki, U. Eisenecker. Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.
- [Dardenne et al. 1993] A. Dardenne, A. Lamsweerde and S. Fickas, "Goal-directed Requirements Acquisition". Science of Computer Programming, Vol. 20(1-2), pp. 3-50, 1993.
- [Dingwall-Smith & Finkelstein 2002] A. Dingwall-Smith and A. Finkelstein, "From Requirements to Monitors by way of Aspects". Early Aspects 2002: Aspect-Oriented Requirements Engineering and Architecture Design, Workshop of the 1st International Conference on Aspect-Oriented Software Development (AOSD), Enschede, The Netherlands, April 2002.
- [Finkelstein & Sommerville 1996] A. Finkelstein and I. Sommerville, "The Viewpoints FAQ". BCS/IEE Software Engineering Journal, Vol. 11(1), pp. 2-4, 1996.
- [France et al. 2004] R. France, D. Kim, S. Ghosh and E. Song, "A UML-Based Pattern Specification Technique". IEEE Transactions on Software Engineering, Vol. 30(3), pp. 193-206, 2004.
- [Georg & France 2002] G. Georg and R. France. "UML Aspect Specification using Role Models". Proceedings of 8th International Conference on Object Oriented Information Systems, Montpellier, France, Lecture Notes in Computer Science, Springer, Vol. 2425, pp. 186-191, September 2002.
- [Grundy 1999] J. Grundy, "Aspect-oriented Requirements Engineering for Component-based Software Systems". Proceedings of the 4th IEEE International Symposium on Requirements Engineering, Limerick, Ireland, IEEE CS Press, pp. 84-91, June 1999.
- [Katz & Rashid 2004] S. Katz and A. Rashid, From aspectual requirements to proof obligations for aspect-oriented systems. In 12th IEEE Int'l Conf. Requirements Engineering (RE), (Kyoto, Japan). IEEE, September 2004.
- [Kazman et al. 1998] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The Architecture Tradeoff Analysis Method", Proc. ICECCS, 1998, IEEE Computer Society Press, pp. 68-78.
- [Kazman et al. 1996] R.Kazman, G.Abowd, L.Bass & P.Clements. Scenario-Based Analysis of Software Architecture. IEEE Software, pp. 47-55, Nov. 1996.

- [Lamsweerde 2001] A. Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour". Proceedings of the 5th International Symposium on Requirements Engineering, Toronto, Canada, IEEE CS Press, pp. 249-261, September 2001.
- [Moreira et al. 2002] A. Moreira, J. Araújo and I. Brito, "Crosscutting Quality Attributes for Requirements Engineering". Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE), Ischia, Italy, ACM Press, pp. 167-174, July 2002.
- [Nuseibeh 2001] B. Nuseibeh, "Weaving Together Requirements and Architectures", IEEE Computer, 34(3), pp. 115-117, 2001.
- [Rashid et al. 2002] A. Rashid, P. Sawyer, A. Moreira and J. Araújo, "Early Aspects: A Model for Aspect-Oriented Requirements Engineering". Proceedings of the IEEE Joint International Conference on Requirements Engineering (RE), Essen, Germany, IEEE CS Press, pp. 199-202, September 2002.
- [Rashid et al. 2003] A. Rashid, A. Moreira and J. Araújo, "Modularisation and Composition of Aspectual Requirements". Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), Boston, USA, ACM Press, pp. 11-20, March 2003.
- [Sommerville & Sawyer 1997] I. Sommerville and P. Sawyer, "Requirements Engineering - A Good Practice Guide". John Wiley and Sons, 1997.
- [Sousa et al. 2004] G. Sousa, S. Soares, P. Borba and J. Castro, "Separation of Crosscutting Concerns from Requirements to Design: Adapting an Use Case Driven Approach", Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design, workshop of the 3rd International Conference on Aspect-Oriented Software Development, Lancaster, UK, 22-26 March 2004.
- [Sutton & Rouvellou 2002] S. M. Sutton Jr. and I. Rouvellou, "Modelling Software Concerns in Cosmos". Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD), Enschede, The Netherlands, ACM Press, pp.127-133, April 2002.
- [Tekinerdogan 2003] B. Tekinerdogan, ASAAM: Aspectual Software Architecture Analysis Method, Early Aspects 2003: Aspect-Oriented Requirements Engineering and Architecture Design, Workshop of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), Boston, USA, March 2003.
- [Whittle & Araújo 2004] J. Whittle, J. Araújo, "Scenario Modeling with Aspects", IEE Proceedings – Software, Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design. Editors: A. Rashid, A. Moreira, B. Tekinerdogan. August 2004.
- [Whittle et al. 2003] J. Whittle, J. Araújo, D. Kim, "Modeling and Validating Interaction Aspects in UML", 4th AOSD Modeling With UML Workshop, workshop of the 6th International Conference on the Unified Modeling Language – <<UML>> 2003, San Francisco, USA, 20 October 2003.
- [Whittle & Schumann 2000] J. Whittle and J. Schumann. "Generating Statechart Designs from Scenarios". Proceedings of the 22nd International Conference on Software Engineering (ICSE), Limerick, Ireland, IEEE CS Press, pp.314-323, June 2000.
- [Yu 1995] E. Yu, "Modelling Strategic Relationships for Process Reengineering". PhD Thesis, University of Toronto, 1995.
- [Yu et al 2004] Y. Yuy, J. Leite, J. Mylopoulos, "From goals to aspects: discovering aspects from requirements goal models", Proceedings of the 12th IEEE International Requirements Engineering Conference (RE), Kyoto, Japan, IEEE CS Press, September 2004.

[Name YR]

Citation goes here.

About the Authors

João Araújo is an assistant professor at the New University of Lisbon, Portugal. He holds a PhD in Computer Science, from the University of Lancaster, United Kingdom, in the area of Requirements Engineering. Currently his main research interest is in aspect-oriented requirements engineering, having several papers on this topic in international conferences and workshops. He was an organizer of the early aspects workshop at AOSD 2002, AOSD 2003, AOSD 2004 and OOPSLA 2004. Also, he was an organizer of two workshops on integration and transformation of UML models (WTUML and WITUML), at ETAPS'01 and ECOOP'02. Additionally, he served on the organization committees of UML'03 as tutorials chair and UML'04 as co-publicity chair. He is also acting as one of the co-publicity chairs of MoDELS 2005. For more information please visit <http://ctp.di.fct.unl.pt/~ja/>

Elisa Baniassad received her PhD in computer science from the University of British Columbia in 2003, and conducted a two-year NSERC post-doctoral fellow at Trinity College, Dublin. She is an assistant professor in the department of Computer Science and Engineering at the Chinese University of Hong Kong. She is interested in empirical studies of developers' practices for finding and dealing with aspects in code. Based on these observations, she works on software-design analysis for elucidation of aspects in system requirements, and establishing traceability links to detailed design and implementation. She is co-author of a book on the identification and design of aspects, to be published by Addison-Wesley. Elisa was co-organiser of the Beyond Design Patterns (mis)Used workshop, held at OOPSLA 2001 and the Early Aspects workshop held at OOPSLA 2004. For more information please visit <http://www.cs.tcd.ie/Elisa.Baniassad>.

Paul Clements is a senior member of the technical staff at Carnegie Mellon University's Software Engineering Institute. There, he works in areas of software architecture and software product lines. He has co-authored four books in software engineering, including three in software architecture, as well as over fifty papers in architecture, documentation, software structure, and product line methodologies. He was the conference chair for the first Working IFIP/IEEE Conference on Software Architecture (WICSA) in 1999, and the 2004 International Software Product Line Conference (SPLC), and has served on the program committee for AOSD. He holds a Ph.D. in computer sciences from the University of Texas at Austin, and a M.Sc. in computer science from the University of North Carolina at Chapel Hill. For more information please visit <http://www.sei.cmu.edu/staff/clements/>.

Ana Moreira is an assistant professor at the Department of Informatics at the Universidade Nova Lisboa, Portugal. She holds a PhD in Computer Science in the area of formal methods and object technology. Her main research areas are object technology, requirements engineering, and formal description techniques. Currently she is interested in investigating how coordination technologies can be used to support agile software evolution and also how aspect-orientation can be used during the early phases of the software development process. She is a co-editor of the upcoming IEE Software Proceedings special issue on Early Aspects and is a member of the editorial board for the Springer-Verlag journals "Software and Systems Modeling" and also the upcoming "Transactions on AOSD". She has been a member of the ECOOP and UML Program Committees for several years and is a member of the pUML (precise UML) steering committee. She was workshop chair for ECOOP'99 and UML'2003, and workshop co-chair for ECOOP'2002. She co-organised several workshops for ECOOP, OOPSLA, AOSD, UML and ETAPS. She is conference chair for UML 2004. For more information please visit <http://ctp.di.fct.unl.pt/~amm/>.

Awais Rashid is a senior lecturer at Computing Department, Lancaster University, UK. He holds a BSc in Electronics Engineering, an MSc in Software Engineering Methods and a PhD in Computer Science. His principal research interests are in aspect-oriented software engineering and aspect-oriented databases. He has been involved in organisation and program committees of several workshops at ECOOP, OOPSLA and TOOLS Europe. He has edited IEE Software Proceedings special issue on Aspect-Oriented and Component-based Software Engineering (2001) and the BCS Computer Journal Special Section on AOP (2003, co-edited with Lynne Blair). He is also a co-editor of the upcoming IEE Software Proceedings special issue on Early Aspects and is the co-editor-in-chief of the journal Transactions on AOSD to be launched shortly by Springer-Verlag. He is coordinating the European Network of Excellence on Aspect-Oriented Software Development. He served on the program committee for AOSD 2002, as the publicity chair for AOSD 2003 and is the organising chair for AOSD 2004. He is a regular presenter at international conferences and workshops and has given a number of tutorials, invited talks and seminars on aspect-orientation and recently published a book on Aspect-Oriented Database Systems. For more information please visit <http://www.comp.lancs.ac.uk/computing/aop/>

Bedir Tekinerdoğan holds an MSc and a PhD degree in Computer Science from the University of Twente, The Netherlands. Currently, he is an assistant professor at University of Twente where he works on the project Aspect-Oriented Software Architecture Design which is carried out with IBM Research, The Netherlands. His current research is basically on aspect-oriented software architecture design, multidimensional separation of concerns using design space modeling and aspect-oriented domain analysis. He served on the program committee and organising committee for AOSD 2002 and was an organiser of the early aspects workshop at AOSD 2002 and AOSD 2003. He was also an organizer of several workshops at ECOOP on the topics of aspect-oriented programming, automating software

development methods, and adaptability in object-oriented software development. From 1994 on, he has been teaching on the topics of aspect-oriented software development, software architecture design, object-oriented analysis and design, and object-oriented software design patterns. For more information please visit <http://www.cs.utwente.nl/~bedir>

REPORT DOCUMENTATION PAGE*Form Approved
OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE February 2005	3. REPORT TYPE AND DATES COVERED Final	
4. TITLE AND SUBTITLE		5. FUNDING NUMBERS F19628-00-C-0003	
6. AUTHOR(S)			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2005-TN-xxx	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116		10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES			
12A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS		12B DISTRIBUTION CODE	
13. ABSTRACT (MAXIMUM 200 WORDS)			
14. SUBJECT TERMS		15. NUMBER OF PAGES	
16. PRICE CODE			
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL