

Towards an Aspect-Oriented Software Development Model with Tractability Mechanism

Mohamad Kassab, Olga Ormandjieva
Department of Computer Science and Software Engineering
Concordia University
1455 de Maisonneuve Blvd West
Montreal, Quebec, H3G 1M8 Canada
{moh_kass, ormandj}@cse.concordia.ca

Abstract

An effective software development approach must harmonize the need to build the functional behavior of a system with the need to clearly model the associated nonfunctional requirements that affect parts of the system or the system as a whole. Aspect-Oriented Software Development (AOSD) aims at providing a systematic support for the identification, separation, representation (through proper modeling and documentation), and composition of crosscutting requirements (both functional and nonfunctional) as well as mechanisms that can make them traceable throughout the software development. In this work, we discuss a sequence of systematic activities towards an early consideration of specifying and separating crosscutting requirements. This approach would make it possible to promote traceability of broadly scoped requirements throughout system development, starting from the Requirements Elicitation phase till the implementation.

1. Introduction

Despite the success of object-orientation (OO) in the effort to achieve separation of concerns, current OO techniques support one dimensional decomposition of the problem focusing on the notion of a class. Such decomposition is not a good candidate to handle the complex interaction of components as it leaves certain properties without being localized in single modular units and as a result their implementation cuts across the decomposition of the system. This is the phenomenon of crosscutting.

The limitation in the modularization techniques that imposes only one way at a time on how the program could be modularized is called the tyranny of the dominant decomposition [1]. Multi-dimensional separation of concerns is aimed at breaking the tyranny to reduce software complexity and improve comprehensibility; promote traceability; facilitate reuse, non-invasive adaptation, customization, and evolution; and simplify component integration.

Crosscutting is not a property of the implementation only but it propagates up to early stages of the software development. The conflict that tends to arise in Object-Oriented Software Development (OOSD), when we map requirements from its N-dimensional domain to the single dimensional solution space constitutes the original source of crosscuttings.

Aspect-Oriented Programming (AOP) is a collective term that refers to a growing family of approaches and technologies that provide better linguistic mechanism for separation of concerns by supplying the process of software development with a second axis of decomposition that enables the identification and separation of core functionality and crosscutting requirements. Implementation of an AOP language seeks to encapsulate crosscutting concerns through the introduction of a new construct called an aspect.

While AOP supports separation of concerns at the code level, Aspect-Oriented Software Development (AOSD) has extended AOP to provide a systematic support for the identification, separation, representation (through proper modeling and documentation), and composition of crosscutting

concerns as well as mechanism that makes them traceable throughout software development.

Adopting aspect-oriented technologies for software development requires revisiting the entire software lifecycle in order to identify and represent occurrences of crosscutting during software requirements engineering and design, and to determine how these requirements are composed. In this paper, our goal is to develop a systematic AOSD model that provides a traceability mechanism for the captured functional requirements (FRs) and nonfunctional requirements (NFRs) throughout the development process. This paper offers the following contributions:

1. It proposes a new approach to identify, separate and compose requirements starting from early requirements elicitation to implementation phase.
2. It provides a new traceability mechanism through the software development process that enables stakeholders from tracing requirements with static and dynamic visions towards the developed software.
3. It provides a new mechanism to compose requirements that assist in integrating the captured main requirements with the crosscutting requirements.

The rest of this paper is organized as follows: In section 2 we will present our AOSD model. In sections 3, 4, 5 and 6 we will describe the phases of the model and the traceability mechanism we propose. In section 7 we conclude and discuss the future research avenues.

2. The AOSD model

Our proposed aspect-oriented model is depicted in Figure 1. The model is composed of five phases: Requirements Elicitation, Analysis and Crosscutting Realization, Composing Requirements, Design and Implementation. We use the term phase to describe a group of one or more activities within the AOSD model. The phase is a mean to categorize activates based on the general target they tend to achieve.

Requirements traceability is provided throughout the model to influence the consistency and change management of the requirements of a system. Requirements traceability is defined as the ability to describe and follow the life of a requirement, in both a forward and backward direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration in any of these

phases) [2]. This is achieved in our model by using two hierarchy graphs to keep track of the required behavior of the system using static and dynamic views of objects starting from requirements elicitation till the implementation. We will refer to the graphs by the static and the dynamic hierarchies.

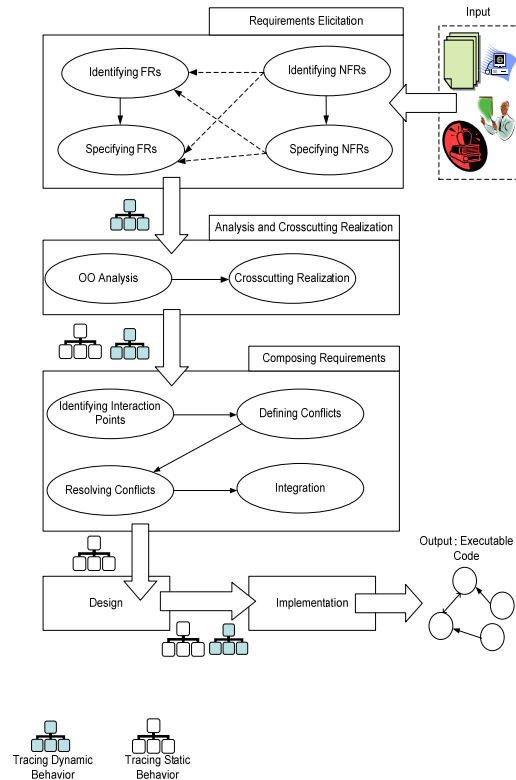


Figure 1: AOSD Model

The hierarchies will be introduced and updated at certain breakpoints within the development process as follows:

1. End of Requirements Elicitation phase: The dynamic hierarchy is introduced. At this phase, we are supposed to have successfully specified the use-cases through scenarios that constitute as the origin for the dynamic behavior of the system.
2. End of Analysis and Crosscutting Realization phase: The static hierarchy is introduced. At this phase, we are supposed to have defined the conceptual classes (through the domain model) that constitute the origin of the static behavior of the system. The dynamic hierarchy is updated to show the effect of crosscutting realization among use-cases.

3. End of Composing Requirements phase: The static hierarchy is updated to show the effect of integrating NFRs with the conceptual classes.
4. End of Design Phase: Both hierarchies are updated to show the extension to the design level through the static artifacts (e.g. class diagram) and dynamic artifacts (e.g. communication diagram).

In the following sections, we will describe each phase of the model in more details and the supporting tracing mechanism.

3. Requirements Elicitation Phase

Requirements Elicitation phase aims to discover the requirements for the system. It is composed of four activities: identifying FRs, specifying FRs, identifying NFRs and specifying NFRS.

Identifying FRs: Functional requirements capture the intended usage of the system. This usage may be expressed as services, tasks or functions which the system is required to perform. The context diagram could be an excellent starting point for capturing the system’s boundaries, users and FRs. Identifying FRs is an activity that involves discussions with stakeholders, reviewing proposals, building prototypes and arranging requirements elicitation meetings.

Specifying FRs: In this activity, we further refine each usage of the system into a detailed functional behavior described as a use-case with textual description. Thus, in this activity, each FR is mapped to one or more use-cases. The outcome is the completion of a use-case description for each use-case (Table 1). Table 1 is similar to the fully dressed format [3].

Use Case No.	Unique to the use-case.
Name	The name of the use-case.
Priority	Importance of the use-case.
Actors	Primary and secondary actors.
Precondition	Textual description of the condition that must be satisfied before the use-case is executed.
Main scenario	A single and complete sequence of steps describing an interaction between a user and a system.
Alternative scenario	Extensions or alternate courses of main scenario.
Postcondition	Textual description of the

	condition that must be satisfied after the use case is executed.
Related Use Cases	Use-cases related to the current use- case.

Identifying NFRs: Nonfunctional requirements that are relevant to the problem domain are captured in parallel to the identification of FRs. While elicitation of NFRs can be accomplished by a number of existing techniques, the most recognized technique is to use NFR catalog [4] where each entry in the catalog is cross listed against the decision of whether it is applicable for the system or not.

Specifying NFRs: Since NFRs often invite many different interpretations from different people, they need to be clarified as much as possible through refinements in discussions with the stakeholders. The stakeholders represent NFRs explicitly as softgoals to be satisfied, i.e., goals to be satisfied not in a clear cut sense but within acceptable limits. The best approach to specify NFRs is by using Softgoal Interdependency Graphs (SIG) [4]. We propose the adoption of a matrix (Table 2) that relates the identified and specified NFRs to the FRs and use-cases they affect. In an invoicing system example, where we have a “Search for Product” functionality that must be provided by the system to the customer in a secure way, security is identified as an NFR that is placed as a constraint on the FR (and eventually on the use-case) “Search”. Specifying the NFR “security” further, will decompose it down into three softgoals: confidentiality, availability and integrity. Depending on the requirements, we could be only concerned with availability as a constraint to be implied on “Search” functionality and thus the corresponding cell in the matrix is to be checked. In the case where an NFR would affect the system as a whole (e.g. portability), all entries in the corresponding column must be checked.

		use-cases				
		NFR ₁	NFR ₂		...	NFR _n
			NFR _{2,1}	NFR _{2,2}		
FR ₁	Use-Case ₁	√				
FR ₂	Use-Case ₂	√	√			
	Use-Case ₃			√		
...	...					
FR _i	Use-Case _n					

From Requirements Elicitation to Analysis: By the end of Requirements Elicitation phase, we are supposed to have successfully managed capturing and specifying FRs and NFRs of the system. Figure 2 introduces the dynamic hierarchy we use to trace the dynamic behavior of the system at the end of this phase.

Each FR is mapped to one or more use-cases. We consider a use-case to be a set of scenarios describing instances of the usage of the system. Each scenario shows the real world concepts (including the system) and the events interchanged between them, ordered in a time sequence. We map the scenarios to sequence of events. High level NFRs are the outcome of Identifying NFRs activity; while low-level NFRs are extracted from the decomposition specified within the SIG. The arrows from the high level NFRs to FRs and from low level NFRs to use-cases are extracted from the dependencies described in Table 2. The arrow signifies that FRs or use-cases are to be provided through the system with the constraints implied by the associated NFRs. Having a high level NFR (e.g. X) been associated with a certain FR (e.g. Y) implies that at least one of the low level NFRs under the hierarchy of X is to be associated with at least one of the use-cases under the hierarchy of Y. It is important to keep in mind that the purpose of this hierarchy is to trace the dynamic behavior of the system and not to model system's requirements. Modeling is to be accomplished within the next activities of the AOSD model.

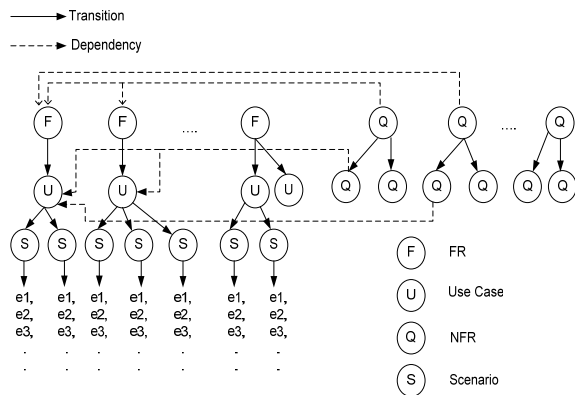


Figure 2: Tracing the dynamic behavior: Requirements Elicitation Level

4. Analysis and Crosscutting Realization

Software requirements analysis is a critical phase of the software development process, as errors at this

stage inevitably lead to later problems in the system design and implementation. In our AOSD model, the analysis phase is composed of two activities: OO Analysis and Crosscutting realization.

OO Analysis: The objective of the OO analysis activity is to understand the textual descriptions (requirements) that have been inducted in previous activities and to abstract the software under development into an OO analysis model. Analysis modeling is the formal or semi-formal presentation of the specification, through which the knowledge and information included in the textual description of the requirements are transmitted to the elements of the OO analysis model. The appropriate elements for OO analysis modeling are: use-case model diagram, System Sequence Diagrams (SSDs), domain model diagram, activity diagram and state charts. In this discussion, we choose to focus on the first three diagrams to present the static and dynamic visions of the system. A domain model represents the static view and it illustrates meaningful (to other modelers) conceptual classes in a problem domain; it is the most important artifact to create during the OO analysis [3]. On other hand, at this activity, we model each sequence of events that are mapped from successful scenarios through an SSD. An SSD treats a system as a black box, placing emphasis on events that cross the system boundary from actors to the system and vice-versa. The set of all required system operations within a SSD is determined by identifying the system events.

Crosscutting Realization: To identify the crosscutting nature of certain use-cases we need to take into consideration the information contained in Related Use Cases row in Table 1. If a use-case is included as a related use-case in several use-cases, then it is crosscutting.

On the other hand, the identified NFRs are classified as crosscuttings as they are considered as global properties of the system and they always crosscut at different spots of it.

In this phase, crosscutting requirements are not modeled. They are only identified. These requirements will be modeled at the integration activity during composing requirements phase.

From Analysis to Composing Requirements: The dynamic hierarchy we proposed before is a good candidate to trace the requirements from a dynamic point of view; and it should be updated by the end of this phase as shown in Figure 3. An arrow between

two use-cases explains that the use-case at the tail of the arrow relies on the use-case at the head of the arrow to be accomplished. In order to trace requirements from a static point of view, we need a second hierarchy similar to one we propose in Figure 4. The diagram is built by relating the conceptual classes modeled in domain model to use-cases they belong to [5]. The diagram also shows that one conceptual class could be shared among different use-cases. To see the relationships among the conceptual classes themselves, we must refer to the domain model diagram.

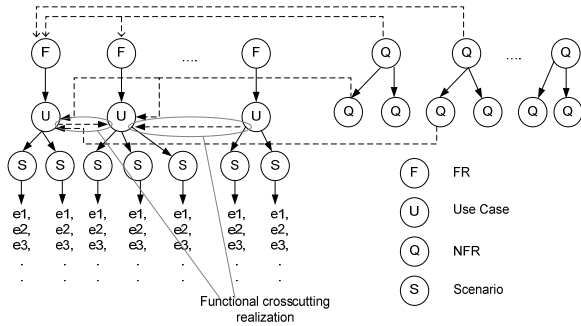


Figure 3: Tracing the dynamic behavior: Analysis and Crosscutting Realization Level

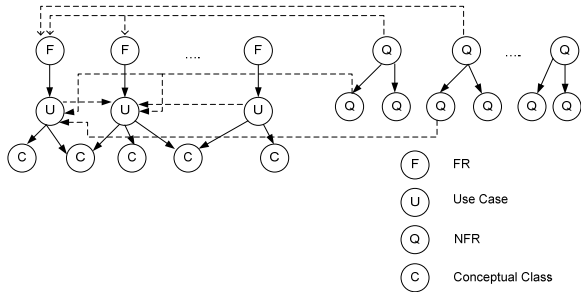


Figure 4: Tracing the static behavior: Analysis and Crosscutting Realization Level

5. Composing Requirements

The goal of composing requirements phase is to integrate identified crosscuttings (both functional and nonfunctional) with the use-case model and the domain model. This is achieved in a series of four activities: (1) identifying the interaction points at which crosscutting requirements affect the system, (2) identifying possible conflicts among requirements at each interaction point, (3) resolving conflicts, and (4) integrating requirements.

Identifying interaction points: Based on requirements crosscuttings, we can identify interaction points in the system where crosscuttings will manifest themselves. We start by defining the set of requirements $R = \{FR\} \cup \{NFR\}$, and the set of crosscuttings $C = \{\text{Crosscutting requirements (CCRs)}\} \subseteq R$. We also define a function A which maps R to sets of CCRs as $A: R \rightarrow P(C)$, where P states for “Powerset”. A tracks those requirements that transverse several other requirements captured by this level of the development cycle.

Let $r \in R$, $c \subseteq C$. We define A as: $A(r) = \emptyset$, if there are no crosscutting requirements at r , and $A(r) = c$, otherwise. The set of Interaction Points I is defined as: $I = R - \{r \mid A(r) = \emptyset\}$.

Defining conflicts: Hardly any requirements manifest in isolation, and normally the provision of one crosscutting requirement may affect the level of provision of another. We refer to this mutual dependency as non-orthogonality. In [6], the authors proposed a symmetric contribution table to show in which way (negatively or positively) an aspect contributes to the others. We use the same table in our model.

Resolving conflicts: For each interaction point $P_i \in I$ we analyze the set $c = A(P_i)$, and study the contribution among its elements. We are essentially interested in those elements (requirements) that have a mutual negative interaction. We manage conflict resolution by assigning priorities of execution of the crosscuttings by mapping $A(P_i)$ to a sequence C_{seq} , where $P_i \in I$. An element in the sequence is either a crosscutting or a set of crosscuttings. The set notation within C_{seq} indicates that the elements within “{ }” are free to execute in any order relative to this position in the sequence, as there is no negative contribution identified. The process of mapping is guided by the expert’s opinion. In the invoicing system example, we could have $A(\text{Place Order}) = \{\text{Availability Confidentiality, Response Time, Process Payment}\}$. But since Confidentiality and Availability have a mutual negative interaction between each other, we have to map the set to a sequence with an assigned order of execution. The result could be similar to $[\text{Confidentiality, Availability, \{Response Time, Process Payment\}}]$. The set element indicates that Response Time and Place Payment are free to execute in any order after Confidentiality and Availability.

Integration: In the integration activity, we compose and model all requirements based on the collected information from previous activities. Currently, there are many proposals in the literature on integrating the concept of aspect with UML [7, 8, 9, etc.]. In our model, we extend the standard UML use-case diagram with a new stereotype <<CCR>> to abstract the crosscuttings integrated into the model, and use the <<include>> relation stereotype to indicate which use-cases are crosscut by the crosscuttings (see Figure 5).

The knowledge required for creating the extended use-case model is extracted from output of function A against each defined requirement. In Figure 5, use-case1, use-case2 $\in I$ and aspect1 $\in (A(\text{use-case1}) \cap A(\text{use-case2}))$.

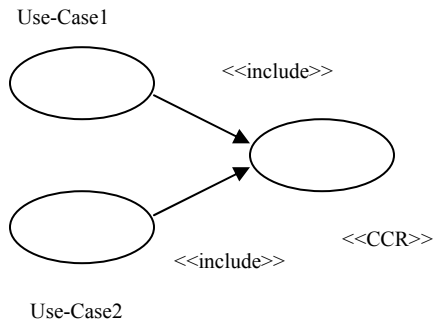


Figure 5: Integrated Use-Case model

We extend domain model to include all NFRs that have been elicited earlier. In Figure 4, we showed how each use-case is mapped to set of concepts. In this activity, we realize that for each NFR (e.g. X) affects a use-case (e.g. Y), X affects at least one defined concept under the hierarchy of Y. We have to define which concepts to be affected by which NFRs without breaking this rule.

From Composing Requirements to Design: The static hierarchy is to be updated by the end of this phase to show the relation between NFRs and the domain's concepts (Figure 6). The dynamic hierarchy will be left with no updates by the end of this phase.

6. Design

For each use-case in the analysis model, we refine further the system operations specified in its SSD into communication diagrams showing the design level details on the interaction between the objects involved in one system operation. Communication diagrams

show the message flow between objects in an OO application and also imply the basic associations (relationships) between classes. Operationalizations are added next to SIG. Operationalization is defined as a possible design solution to satisfy the requirement [3]. In case the operationalization is chosen to be a method to be implemented then we have to define:

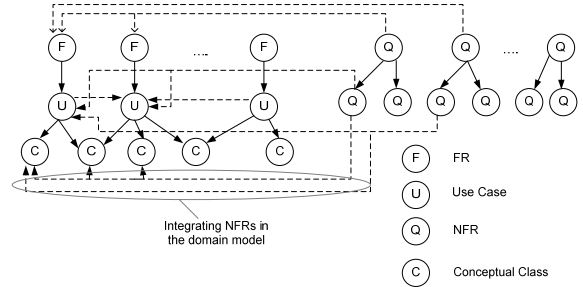


Figure 6: Tracing the static behavior: Analysis and Crosscutting Realization Level

1. The communication diagrams at which the operationalization will be involved.
2. Points of communication between the operationalization and objects in each communication diagram.
3. Semantics of communication between the Operationalization and objects in each communication diagram (before, after or around). We refer to this semantics as composition operators. If two operationalizations are intended to interact at the same point with the same composition operator, then we have to assign priorities to avoid a direct conflict. Priorities could be assigned based on c_{seq} defined earlier. If two operationalizations are defined for the same NFR are to contribute at the same communication point (message) with same composition operator, then a further discussion with stakeholders is required to re-assign priorities.

On other hand, if two use-cases (e.g. X and Y) having a common related use-case (e.g. Z), then it is of high probability (but not necessary true) that a common messages exchanged within communication diagrams defined under the hierarchy of use-case Z exist as common messages exchanged within communication diagrams defined under the hierarchy of use-cases X and Y. Those common messages will be recognized in communication diagrams as they present a form of crosscuttings. In [10], we proposed to present operationalizations and common messages

generated out of a common functional behavior within communication diagrams using special notation: \otimes

Class diagram is built next using the domain model, and the composed communication diagrams.

From Design to Implementation

We choose to map the design components to (Aspect J) code. The hierarchy diagrams used to trace the static and dynamic vision of the system are further extended to look similar to what we propose in Figures 7 and 8 respectively. If an NFR is affecting a use-case then one or more of the operationalizations defined under the hierarchy of that NFR will be affecting the messages defined under the hierarchy of the use-case. We are ready to map the design to implementation using the following rules:

1. For each class defined in the class diagram, it will be mapped to a class in the implementation.
2. For each operationalization that appears in a communication diagram, it will be mapped to an aspect.
3. For each common message within two or more collaboration diagrams recognized out of a common functional behavior, it will be mapped to an aspect.
4. For each rule in the design that defines at which point in a communication diagram an operationalization or a common message will be involved (point of communication), it will be mapped to a joinpoint.
5. For each rule in the design that defines the composition operator for an operationalization or a common message, it will be mapped to an advice.

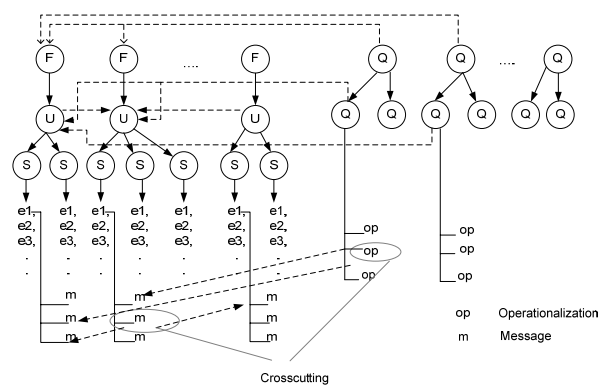


Figure 7: Tracing the dynamic behavior: Design level

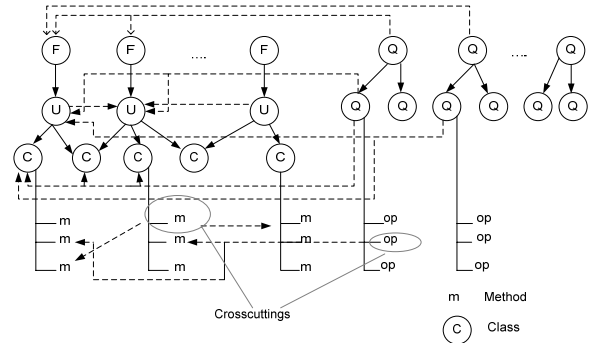


Figure 8: Tracing the static behavior: Design level

7. Conclusion

Tangling and scattering are symptoms that do not exclusively affect implementation, but they also propagate to early stages of the development process. Identifying and modeling crosscuttings earlier has a great impact on improving the general quality of the system and reducing complexity by (1) prompting understandability and reusability, (2) enhancing the process of detecting and removing defects, and (3) reducing development time.

In this paper, we discussed a sequence of systematic activities towards an early consideration of identifying, specifying and separating broadly scoped requirements that are traceable throughout system development process. We addressed both FRs and NFRs as candidate crosscutting requirements. We provided a tracing mechanism to track the system under the development from two views: the static and the dynamic behaviors. For future research, we plan to investigate how to formalize the AOSD model and the tractability mechanism.

8. References

- [1] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton, "N Degree of Separation of Concerns". In Proceedings of the 21st International Conference on Software Engineering, pages 107–119, Los Angeles, CA, USA, 1999.
- [2] Gotel, Orlena, "Contribution Structures for Requirements Traceability", London, England: Imperial College, Department of Computing, 1995.
- [3] C. Larman, "Applying UML and Patterns; An Introduction to Object-Oriented Analysis and Design and the Unified Process", 3rd edition. Upper Saddle River, NJ: Prentice Hall Inc., 2004.

- [4] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, "Nonfunctional Requirements in Software Engineering", Kluwer Academic Publishing, 2000.
- [5] K. Meridji, O. Ormandjieva, "Measuring Consistency of the Analysis Model: An XML Approach", In Proceedings of the 13th International Workshop on Software Measurement (IWSM2003), Montreal, QC, 2003.
- [6] A. Rashid, A. Moreira, and J. Araujo, "Modularisation and Composition of Aspectual Requirements", In 2nd International Conference on Aspect-Oriented, pages 11–20, Boston, MA, 2003.
- [7] A. Moreira, J. Araujo, and I. Brito, "Crosscutting Quality Attributes for Requirements Engineering", In 14th International Conference on Software Engineering and Knowledge Engineering, pages 167–174, Ischia, Italy, 2002.
- [8] D. Park and S. Kand, "Design Phase Analysis of Software Performance Using Aspect-Oriented Programming", In 5th Aspect-Oriented Modeling Workshop in Conjunction with UML 2004, Lisbon, Portugal, 2004.
- [9] J. Araujo, A. Moreira, I. Brito, and A. Rashid, "Aspect-Oriented Requirements With UML in Conjunction with 1st International Conference on Aspect-Oriented Software Development", In Workshop on Early Aspects in Conjunction with 3rd International Conference on Aspect Oriented Software Development, Enschede, Netherlands, 2002.
- [10] Mohamad Kassab, Constantinos Constantinides, and Olga Ormandjieva. Specifying and Separating Concerns From Requirements to Design: a Case Study. In The IASTED International Conference on Software Engineering (ACIT-SE 2005), pages 18–27, Novosibirsk, Russia, 2005.