Applying Aspect-Orientation in Designing Security Systems: A Case Study

Shu Gao, Yi Deng, Huiqun Yu, Xudong He, Konstantin Beznosov[i], Kendra Cooper[ii]
School of Computer Science, Florida International University
[i]Department of Electrical and Computer Engineering, Univeristy of British Columbia
[ii]Department of Computer Science, University of Texas at Dallas
{sgao01, deng, yhq, hex}@cs.fiu.edu; [i]beznosov@ece.ubc.ca; [ii]kcooper@utdallas.edu

Abstract. *As a security policy model evolves, the design of security systems using that model could become increasingly complicated. It is necessary to come up with an approach to guide the development, reuse and evolution of the design. In this paper, we propose an aspect-oriented design approach to designing flexible and extensible security systems. A case study demonstrates that such an approach has multifold benefits and is worth further exploration.*

1. Introduction

A security policy model always evolves; accordingly, the design of a security system using that policy model should reflect the changes. Using role-based access control (RBAC) as an example, currently it supports role hierarchy, static separation of duty relations, and dynamic separation of duty relations. As research on RBAC progresses, more concerns have been and will be covered. So the model hierarchy of RBAC is quickly becoming more and more complicated, which requires that the security system supporting RBAC be flexible and extensible. To address this issue at the design level, we propose an aspect-oriented approach to designing flexible and extensible security systems. This paper illustrates the approach through a case study, which is part of a design for CORBA access control (AC) supporting RBAC models.

Although some papers in the literature have dealt with separating security concerns in application system design, little research has been done to explore the use of aspect-orientation in designing security systems. Our work is a first step toward a systematic aspect-oriented approach to advance the design of security systems.

2. A Case Study

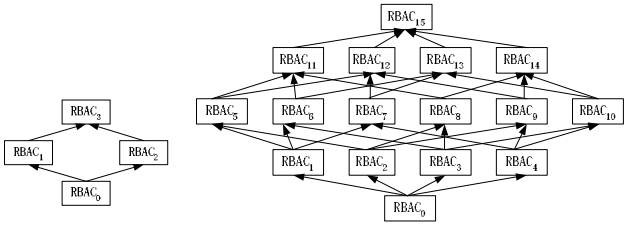
The CORBA AC [13] is a reference model for enforcing access control in the middleware layer of distributed applications. It is aimed to provide a standard way to separate access control and application logic. CORBA AC specification is policy neutral in that only essential and general access control interfaces are specified. To implement a functional CORBA AC mechanism, certain access control policy models have to be supported. In this case study, we choose RBAC models, which have been widely recognized as a well-defined general approach for access control in large-scale authorization management.

2.1. Problem Analysis

In [14], the RBAC96 family contains four models: $RBAC_0$, $RBAC_1$, $RBAC_2$, and $RBAC_3$. $RBAC_0$ is the base model that contains (1) entities – users (U), roles (R), permissions (P); (2) static relationships – user assignment (UA – between users and roles), permission assignment (PA – a between roles and permissions); and (3) dynamic relationship – sessions (S) (a one to many relationship between a single user and his/her multiple roles). $RBAC_1$ extends $RBAC_0$ with a hierarchical structure representing the partial order relation on roles. $RBAC_2$ extends $RBAC_0$ with constraints on entities such as conflicting roles as well as relationships such as a user can only assume a limited number of roles. $RBAC_3$ is the combination of both extensions of hierarchy and constraints such that constraints can be defined on roles at the different levels of the hierarchy.

Since $RBAC_1$ to $RBAC_3$ are derived from $RBAC_0$, one design issue is how to effectively reuse the design for $RBAC_0$ to realize $RBAC_1$ to $RBAC_3$. The RBAC family is still evolving. The number of RBAC models is increasing to cover a variety of emerging concerns and specific application needs. For example, in the proposed RBAC standard by NIST [4], the time concern is incorporated into the concept of dynamic separation of duty relations (DSD), while the old constraint model was called static separation of duty relations (SSD). Very likely, context concern will also be introduced in the near future. If we follow the conventions used in [14], we can illustrate the evolution of RBAC family with Figure 1.

In Figure 1.b, $RBAC_3$ is a new model with temporal constraints (DSD)[1]; $RBAC_4$ is yet another new model covering context (spatial) concern. It is remarkable how fast the complexity can grow with the introduction of new concerns. Hence another very important design issue is how to achieve flexibility and extensibility in designing security systems using such models.



a. The RBAC96 Model Hierarchy

b. The RBAC Model Hierarchy with Time and Context Concerns

Figure 1. Evalution of RBAC family

2.2. Design Approach

Given the above issues, it is necessary to have a design approach that facilitates design reuse and evolution. Separation of concerns [5] has been one of the fundamental principles in software development in the past three decades. At design phase, separation of concerns allows designers to focus on one concern without being distracted by other complexities. In our case study, following this principle can help us manage complexity, comprehensibility, composition and evolution of the design.

Recently, a new software implementation paradigm called aspect-oriented programming (AOP) based on the principle of separation of concerns was proposed [7], which has generated extensive research interest. As Kiczales et al. point out in [7], existing programming languages including procedural, functional, and object-oriented languages decompose a system into functional components. However the implementations of some properties (e.g. synchronization, real-time constraints, error handling, audit, security enforcement) cannot be encapsulated into a single component. Frequently classified as "crosscutting properties", these properties are usually present in more than one functional component. Implementations of such properties in mainstream languages necessarily result in tangled code. Code tangling denotes the use of a single method to implement multiple properties. The purpose of AOP is to provide mechanisms that explicitly capture crosscutting structures, so crosscutting concerns can be encapsulated.

[1] The $RBAC_3$ in RBAC96 family is now $RBAC_5$ in the extended RBAC family.

The studies in AOP have already been extended to aspect-oriented design (AOD), due to the significance of software architecture in system development. In order to obtain a good aspect-oriented design, three key issues must be addressed:

(1) The identification of aspects;
(2) The notations used to specify aspects;
(3) The rules to compose aspects together.

Yet another important issue is the analysis method of the design product. But this is beyond the scope of this paper.

For this case study, we regard each concern in RBAC models as an aspect and thus we have four aspects: role hierarchy (RH), static constraints (SSD), temporal constraints (DSD), and spatial constraints (SC). These four aspects are orthogonal and are faithful reflections of the separation of concerns principle. With this aspect-oriented view, the development of RBAC models will be incremental and compositional. For example, $RBAC_{13}$ (Figure 1.b) will be built by integrating the base model $RBAC_0$ with aspects RH, DSD, and SC. Therefore this approach will greatly enhance the reusability of the base model and aspects, as well as provide great flexibility for RBAC evolution to meet new system needs. Thus we have a nice and elegant solution to issue (1).

A common practice in AOD is to extend UML notations [6] as AOD notations. The benefit of using UML is the ease of learning and use. Issue (3) is usually closely related to the implementation models. Our proposed aspect-oriented approach is flexible in that it does not depend on any particular implementation model. For the CORBA AC design, we use the widely studied AspectJ [1] as the implementation model. Consequently, the composition rules of AspectJ are adopted. In the following subsection, we briefly introduce AspectJ and the extended UML notations to be used in our design.

2.3. AspectJ and UML Extension

AspectJ is an aspect-oriented extension of Java. AspectJ defines two types of crosscutting: dynamic crosscutting and static crosscutting. Dynamic crosscutting supports defining and advising points during the dynamic execution of a program. Static crosscutting allows adding new attributes, operations, and many other declarations that may affect the static type hierarchy to a class or aspect. By explicitly capturing dynamic and static crosscutting, AspectJ provides a totally new way to encapsulate crosscutting concerns. Novel as it is, the aspect-oriented method behind AspectJ is relatively easy to understand. Some key concepts are defined (from [8], modified) as below:

Join point: A predictable point in the execution of an application.

Pointcut: A structure designed to identify and select join points within a program.
Advice: Code to be executed when a join point is reached in the application code.
Inter-type declaration: A powerful mechanism to add attributes and methods to previously established classes.
Aspect: A structure analogous to an object-oriented class that encapsulates join points, pointcuts, advices, and inter-type declarations.

Join point, pointcut, and advice are used to realize dynamic crosscutting. The join point is a well-defined point in a program where another concern will crosscut this program. It can be method calls, constructor calls, method call execution, constructor call execution, field get, field set, exception handler execution and other points in the execution of a program. AspectJ uses a designator that takes a join point as a parameter to tell the aspect-oriented program when it should match the join point. The pointcut is a structure to group such designators. Whenever a join point is matched by a designator, the pointcut containing it is triggered. Some advice defined for the triggered pointcut will be executed. Depending on the type of the advice (before, after or around), the code in the advice is executed before, after, or in place of the join point. Inter-type declaration is for static crosscutting. New attributes and methods can be added to existing classes without having to explicitly modify the classes. AOP introduces a new component type – aspect. The aspect is used to encapsulate crosscutting concerns. It contains the join points, pointcuts, and advices.

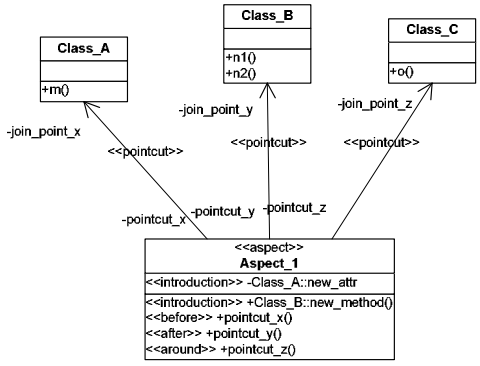


Figure 2. Extension of UML class diagram

We informally extend UML notations to model aspect-oriented design (Figure 2)[2]. An aspect is a regular class with the newly created stereotype <<aspect>>. An inter-type declaration has a new stereotype <<introduction>>. It is like an attribute or a method in a regular class, except that its name should start with the name of the target class/aspect to which the new attribute/method is introduced. Advices have the stereotypes of <<before>>, <<after>> and <<around>>. An advice has no name. The name after <<before>>, <<after>> or <<around>> is the name of the pointcut for which an advice is defined. A pointcut is represented by one or more navigated association(s) from an aspect to a class/aspect which the aspect crosscuts. The pointcut's name is labelled at the crosscutting aspect side. The join point's name is labelled at the side of the class/aspect being crosscut.

2.4. The Aspect-Oriented Design

Based on the above discussion, this subsection introduces an aspect-oriented design for CORBA AC that operates with $RBAC_0$-$_3$ in the RBAC96 family. It is not our purpose to present a complete and detailed design here; instead, we would focus on demonstrating how AOD realizes the separation of concerns principle, and how it helps to manage the complexity shown in Figure 1.

Base Design – Main Concern

As we have analyzed in subsection 2.1 and 2.2, the main concern of this case study is to realize a CORBA AC mechanism that supports $RBAC_0$. The design of the main concern will be reused and crosscut by the design of new concerns, therefore it is called the base design. When working on a design, it is better to have some knowledge of other concerns that may arise. However, it is always the case that the designers hardly know what will happen in the future. The good news is that, with AOD, we do not have to worry about other concerns.

Aspect One – Role Hierarchy

Let us see what new attributes and methods need to be introduced and which existing methods need to be modified to support role hierarchy. First, as a direct result of role hierarchy, functions used to manage the partial order relation are need: add_inheritance(), delete_inheritance(). They should be added to the Role class in the base design. Consequently, the Role class needs to maintain a list of immediate ascendants and a list of immediate descendants. Second, in the base design, there is a method get_assigned_roles(user) in the UAList class, which returns all roles assigned to the given user and is used to determine a user's access permission to resources. When role hierarchy exists, get_assigned_roles(user) cannot return all roles that a user actually has, since some roles not assigned can be inherited. For example, in a bank, the role manager inherits the role employee. If John is assigned to be the manager, then he is also a bank employee though he is not explicitly assigned to that role. The access control system needs to find all roles a user actually has in order to determine the user's permissions correctly. Therefore, we add get_authorized_roles(user) to the UAList class for returning all roles including the inherited ones of a user.

[2] Some ideas are borrowed from [15].

Similarly, we need authorized_users(role) (in the UAList) and authorized_roles(user) (in the UA class) to take the place of corresponding "assigned_" ones in the base design. Accordingly, in the base design, two methods that used to call get_assigned_roles(user): authenticate() from the PrincipalAuthenticator class and set_roles() from the Credentials class, now have to been modified to call get_authorized_roles(user).

The concern to support role hierarchy crosscuts the main concern in that it cannot be implemented in a localized way with vanilla object-oriented approach (Figure 3). Several classes in the base design need to be modified or extended. On one hand, the crosscutting problem makes it expensive to modify; on the other hand, the resulting design is hard to understand and maintain.

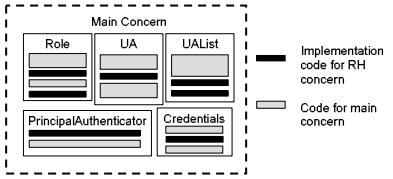


Figure 3. Tangled implementation of RH concern

With AOD, we can address this problem by explicitly representing crosscutting, and encapsulate the crosscutting concerns into aspects. The AOD class diagram for implementing $RBAC_1$ is shown in Figure 4. In the figure, two dashed frames are used to indicate the design for the main concern and the design for the role hierarchy concern respectively. Since the base design is too large, only those classes directly affected by adding the new concern are listed here and relationships other than crosscutting are omitted. As it shows, the

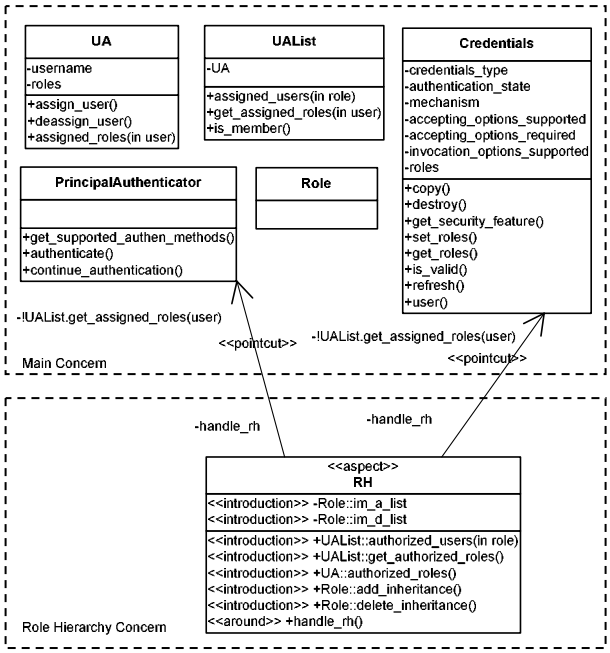


Figure 4. AOD for implementing $RBAC_1$

implementations of two concerns are well modularized without any tangling. An aspect called RH contains all the implementation of the RH concern. Inside the RH aspect, several inter-type declarations are defined to insert new attributes and methods into existing classes. Only one pointcut handle_rh and one join point !UAList.get_assigned_roles(user) ("!" means it is a method call type join point) are defined. At runtime, any method call to UAList.get_assigned_roles(user) generated by PrincipalAuthenticator or Credentials instance will trigger the handle_rh pointcut. The <<around>> type advice code defined for handle_rh will then be executed in place of the UAList.get_assigned_roles(user) method. In this design, the advice code will call UAList.get_authorized_roles(user) which is defined in the same aspect.

Aspect Two – Static Constraints

$RBAC_2$ allows security administrator to set static separation of duty constraints on the assignment of users to roles. In [4], an SSD constraint is defined in the form of (*rs, n*) where *rs* is a role set, and *n* is called "cardinality" which is a natural number ≥ *2*. (*rs, n*) means that no user is assigned to *n* or more roles from the set *rs*.

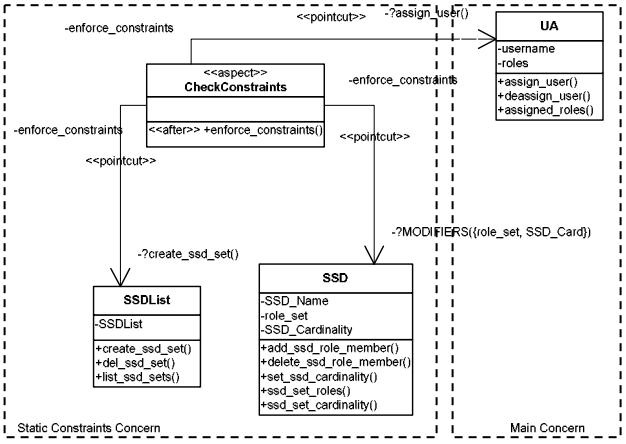


Figure 5. AOD for implementing RBAC2

To implement $RBAC_2$, first we need several functions to manage SSD constraints. They are: create_ssd_set(), add_ssd_role_member(), del_ssd_role_member(), del_ssd_set(), set_ssd_cardinality(), list_ssd_sets(), ssd_set_roles(), and ssd_set_cardinality(). Besides these, every time the SSD relation or the user-role assignment relation is modified, the system must check whether the SSD constraints have been broken. So there should be a function to enforce these constraints.

It is worth noticing that the management functions for SSD constraints do not crosscut the base design. They are newly defined functions and do not need to be inserted into any classes in the base design. Should they be encapsulated into an aspect structure? We prefer not,

since we can define two new classes: SSD and SSDList, which can encapsulate these functions quite well.

The implementation of $RBAC_2$ crosscuts the main concern only at the point where assign_user() of the UA class is executed. A method call to the function that enforces SSD constraints need to be added after the execution of assign_user().

The function enforcing SSD constraints crosscuts SSD and SSDList class, because these two classes contain methods that may change the SSD relation.

Thus, we design an aspect CheckConstraints. In this aspect, there is a pointcut enforce_constraints. An <<after>> type advice is defined for this pointcut. Inside the advice is the code enforcing SSD constraints. There are several join points defined. All of them are of method call execution type (which will be represented by "?" in the diagram). Specifically, the execution of SSDList.create_ssd_set(), UA.assign_user(), and any methods in SSD class that modifies the role_set or SSD_Cardinality attribute will trigger the enforce_constraints pointcut.

The aspect-oriented design for static constraints concern is shown in Figure 5. Although the static constrains concern is not implemented by one aspect, but by two classes and an aspect, the implementation of this concern is still well modularized.

Composition Design – $RBAC_3$

$RBAC_3$ combines role hierarchy and static constraints. Now the advantage of AOD is obvious. By composing the base design, Aspect One and Aspect Two together, with minor modification and without destroying current modularity, we get the design for $RBAC_3$ (Figure 6). According to the composition rule of AspectJ, the aspect RH dynamically crosscuts the aspect CheckConstraints. This is because the advice code enforcing SSD constraints used to call get_assigned_roles(user) to find a user's roles. With the existence of role hierarchy, now get_assigned_roles(user) should be replaced by get_authorized_roles(user). We also need to define a new join point, which is the execution of Role.add_inheritance(). It will trigger the enforce_constraints pointcut. In the figure, two <<pointcut>> associations from RH to CheckConstraints and from CheckConstraints to Role reflect these modifications.

3. Related Work

Aspect-oriented programming is an emerging technology. Recently the research on how to extend this paradigm to design level has attracted more and more attention [3, 16, 17]. The application of AOD to security domain is promising. However, research results are rare. Both [2] and [9] point out that the separation of concerns principle can be used to separate security concerns from application concerns. This is an important and relatively obvious application of AOD to security. Unlike them, we explore the use of aspect-orientation to advance the design of security systems. Due to the novelty of AOD, virtually no research has been done in this direction.

A number of UML extensions have been proposed to support AOD. Examples of such extensions are [11, 12,

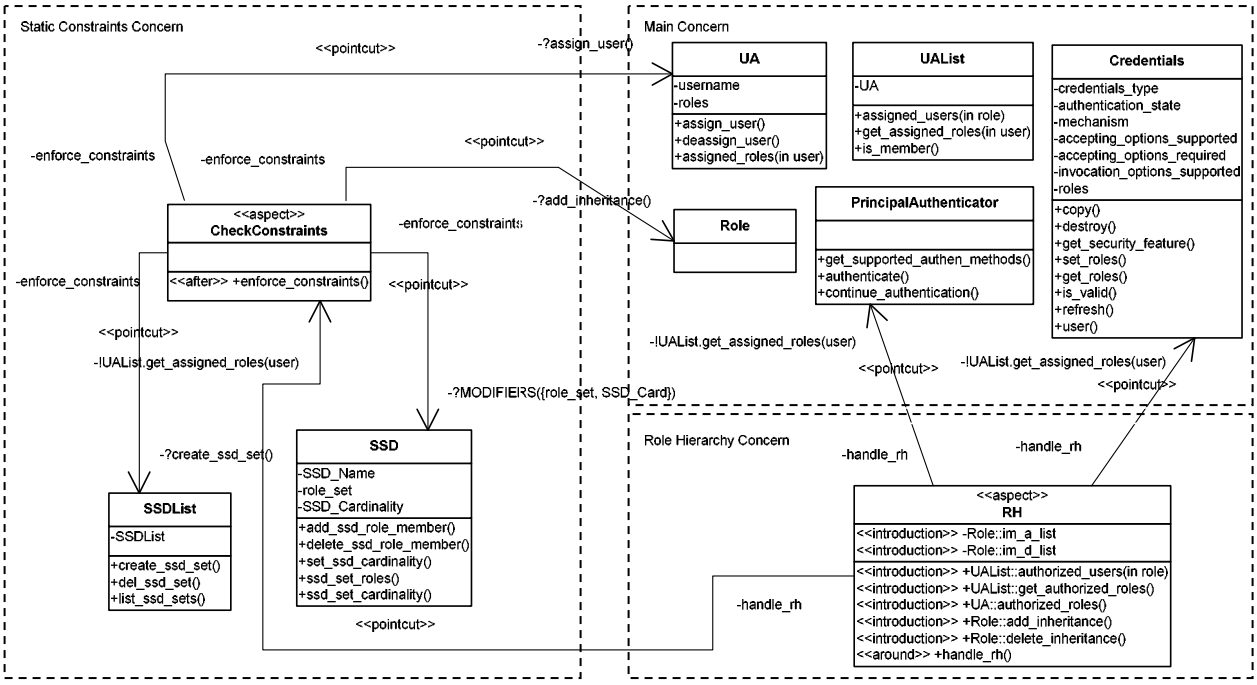


Figure 6. AOD for implementing $RBAC_3$

15]. So far, no extension has been widely accepted. This to some extent hampers the application of AOD. Based on the belief that UML notation should be easy to read and understand, we introduced some stereotypes with [15] as an aid for describing the CORBA AC design.

There is little work reported on implementing RBAC in CORBA systems. The design in this paper is based on our previous research, described in [10], which shows that CORBA Security architecture is capable of supporting $RBAC_0$ – $RBAC_3$ and determines strategies for implementation. However, it does not propose a specific design of CORBA Security. Using one of the strategies from [10], this paper suggests a specific way for implementing RBAC96 model on CORBA systems.

4. Conclusion

The principle behind AOD is separation of concerns. By applying AOD approach in CORBA AC design, a number of benefits of separation of concerns are acquired. Since RBAC extensions covering different concerns can be encapsulated using aspects, we get better modularity with the CORBA AC design. Better modularity leads to better comprehensibility, reusability, flexibility and maintainability. Because there are well defined mechanisms explicitly supporting both dynamic and static crosscutting, the design can be incrementally extended to cover temporal, spatial or other future concerns in RBAC models.

Through this case study, we propose an aspect-oriented design approach to designing security systems. Our work is a first step toward a systematic aspect-oriented approach to advance the design of security systems. Our approach is easy to learn and apply. Although we have used the composition rules of AspectJ and an extended UML design notation for the design presented, our approach does not depend on a specific implementation model.

Our next step is to apply formal methods in AOD. Formal analysis is very useful for detecting possible errors early in the design phase, which is especially important to the design of security systems.

5. Acknowledgements

This work is supported in part by NSF under grant No. CCR-0226763 and No. HRD-0317692.

References

[1] AspectJ homepage. http://eclipse.org/aspectj/

[2] B.D. Win, F. Piessens, W. Joosen and T. Verhanneman. On the Importance of the Separation-of-Concerns Principle in Secure Software Engineering. In Workshop on the Application of Engineering Principles to System Security Design, December 22, 2002.

[3] B. Tekinerdogan and M. Aksit. Deriving Design Aspects from Canonical Models. In Object-Oriented Technology, S. Demeyer and J. Bosch (Eds.), LNCS 1543, ECOOP'98 Workshop Reader, Springer Verlag, pp. 410-413, July 1998.

[4] D.F. Ferraiolo, R. Sandhu, S. Gavrila, D.R. Kuhn and R. Chandramouli. Proposed NIST Standard for Role-Based Access Control. ACM Transactions on Information and System Security, vol. 4, pp. 224-274, 2001.

[5] E.W. Dijkstra. A Discipline of Programming. Englewood Cliffs, NJ: Prentice Hall, 1976.

[6] G. Booch, J. Rumbaugh and I. Jacobson. The Unified Modeling Language User Guide. Addison-Wesley Longman, Inc, 1999.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier and J. Irwin. Aspect-Oriented Programming. In Proceedings of ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyvaskyla, Finland, 1997.

[8] J.D. Gradecki and N. Lesiecki. Mastering AspectJ: Aspect-Orineted Programming in Java. Wiley Publishing, Inc, 2003.

[9] J. Viega, J.T. Bloch and P. Chandra. Applying Aspect-Oriented Programming to Security. Cutter IT Journal, vol. 14, no. 2, pp. 31-39, 2001.

[10] K. Beznosov and Y. Deng. A Framework for Implementing Role-Based Access Control Using CORBA Security Service. In the Fourth ACM Workshop on Role-Based Access Control, Fairfax, Virginia, USA, Octorber, 1999.

[11] M. Basch and A. Sanchez. Incorporating Aspects into the UML. In Proceedings of Third International Workshop on Aspect-Oriented Modeling, March 2003.

[12] O. Aldawud, T. Elrad and A. Bader. UML Profile for Aspect-Oriented Software Development. In Proceedings of Third International Workshop on Aspect-Oriented Modeling, March 2003.

[13] OMG. CORBA Security Service Specification, Version 1.8, March 2002.

[14] R. Sandhu, E. Coyne, H. Feinstein and C. Youman. Role-Based Access Control Models. IEEE Computer, 29(2):38-47, February 1996.

[15] R. Pawlak, L. Duchien, G. Florin, F. Legond-Aubry, L. Seinturier and L. Martelli. A UML Notation for Aspect-Oriented Software Design. In Aspect-Oriented Modeling with UML Workshop at AOSD 2002, Enschede, the Netherlands, 2002.

[16] S. Clarke and R. J. Walker. Composition Patterns: An Approach to Designing Reusable Aspects. In Proceedings of the 23rd International Conference on Software Engineering (ICSE), Toronto, Canada, May 2001.

[17] S. M. Sutton Jr. and P. Tarr. Aspect-Oriented Design Needs Concern Modeling. In Aspect Oriented Design 2002 Workshop, April 23, Enschede, The Netherlands.