Describing Access Control Models as Design Patterns Using Roles

Dae-Kyoo Kim, Pooja Mehta, Priya Gokhale
Department of Computer Science and Engineering
Oakland University
Rochester, MI 48309
{kim2,pmehta,pvgokhal}@oakland.edu

ABSTRACT

An access control model describes at a high level of abstraction a mechanism for governing access to shared resources. In this paper, we view an access control model as a design pattern providing a general solution for ensuring confidentiality, integrity and availability of information resources. We present three widely used access control models, DAC, MAC and RBAC as design patterns using the POSA template. We use an extension of the UML to represent the structure and behavior of the patterns. The extension enables capturing variations of pattern instances. We also attempt to give more details on the problem domain of the patterns to help pattern selection.

1. INTRODUCTION

An access control model is an abstraction of an access control mechanism which enforces access control policies specifying who can access what information under what circumstances. There are many access control models which can be categorized into Discretionary Access Control (DAC) [13], Mandatory Access Control (MAC) [24] and Role-Based Access Control (RBAC) [9]. DAC models enforce access control based on user identities, object ownership and permission delegation. The owner of an object may delegate the permission of the object to another user. MAC models govern access based on the sensitivity level of subjects and objects. A subject may read an object if the security level of the subject is higher than that of the object. RBAC models enforce access control based on roles. Accessibility is determined by the permissions and users assigned to roles.

We view an access control model as a design pattern providing a general solution to a class of access control problems concerning the confidentiality, integrity and availability of information resources in software systems. There has been a huge volume of literature (e.g., [5, 6, 9, 13, 24]) that describes access control models. However, there is only little work describing access control models as design patterns in a pattern template. Existing pattern descriptions of access control models (e.g., see [7]) use a typical example to describe the structure and behavior of an access control model. While use of a typical example is useful to describe an access control model conceptually, how the model may vary in different applications is not captured, which is an important aspect from a practitioner's point of view.

In this paper, we present pattern descriptions for three widely used access control models, each for DAC, MAC and RBAC using the Role-Based Metamodeling Language (RBML) which is a UML-based pattern specification language developed in our previous work [10, 14]. Unlike other pattern descriptions where pattern structure and behavior are described by a typical instance e.g., see [4, 11, 25]), the presented pattern descriptions capture pattern variations with the RBML. The RBML describes a pattern in terms of roles [16] where a role can be played by multiple model elements (e.g., classes). The properties of a role constrain the eligibility of model elements to play the role. We also attempt to give more details on the problem domain of the patterns. We found that most pattern descriptions mainly focused on the pattern's solution domain, and little attention is paid to the problem domain of patterns which help to determine the applicability of a pattern for a given problem. As such, from a pattern user's point of view, problem domains are as important as solution domains. We use the pattern-oriented software architecture (POSA) template [4] to present the DAC, MAC and RBAC patterns.

The rest of the paper is organized as follows. Section 2 gives an overview of the RBML, Section 3 presents the descriptions of the DAC, MAC and RBAC patterns, and Section 4 concludes the paper.

2. THE ROLE-BASED METAMODELING LANGUAGE (RBML)

We use the Role-Based Metamodeling Language (RBML) [10, 14] to describe the structure and behavior of an access control pattern. The RBML is a UML-based pattern specification language that defines a design pattern in terms of roles [16] where a role is played by model elements (e.g., classes). A role has a base metaclass in the UML and only the instances of the base metaclass can play the role. A role can be played by multiple model elements which enables capturing various pattern instances. Fig. 1 shows the relationships between a RBML role and the UML infrastructure [30].

In the figure, the *Item* class role, which is denoted by "|" symbol, is defined at the metamodel-level and has the *Class*

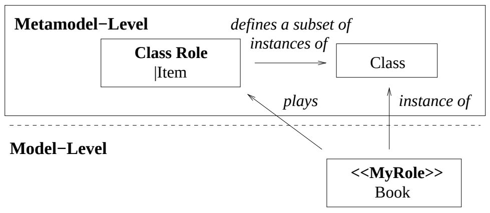


Figure 1: Relationship between Model Role and UML Infrastructure

metaclass in the UML as its base (denoted above the role name). That is, only the instances of the *Class* metaclass can play the role. For example, the *Book* class in the figure is an instance of the *Class* metaclass, and thus can play the *Item* role. In this paper, we assume that all classifier roles have the *Class* metaclass as the base, and as such the base is not explicitly specified. Not every instance can play the role. A role defines a set of constraints on the base metaclass, and only those that satisfy the constraints can play the role. There are two types of constraints, *metamodel-level constraints* and *constraint templates*. Metamodel-level constraints define well-formedness rules on the base metaclass, and model-level constraints define model-level properties such as pre and postconditions and invariants that a model element playing the role must satisfy. Every role has a realization multiplicity which is a metamodel-level constraint for constraining the number of elements that can play the role. If the realization multiplicity is not specified, the default multiplicity *1..** is used specifying that there must be at least one element playing the role. In this paper, we only consider metamodel constraints of base metaclasses and realization multiplications which are sufficient to capture structural variations of a design pattern. RBML roles should not be confused with object roles in the UML [30] which are played by objects.

The RBML provides three types of specifications, *Static Pattern Specifications* (SPSs) capturing the structural properties of a pattern, *Interaction Pattern Specifications* (IPSs) capturing the interaction of pattern participants and *Statemachine Pattern Specifications* (SMPSs) capturing state-based pattern behavior [15]. In this paper, we use only SPSs and IPSs.

An SPS characterizes the structural aspects of a design pattern in a class diagram view. An SPS consists of *classifier* and *relationship* roles whose bases are the *Classifier* and *Relationship* metaclasses in the UML metamodel. A classifier role is associated with a set of feature roles that determines the characteristics of the classifier role. A classifier is connected to other classifier roles by relationship roles. A class diagram is said to conform to an SPS if the class diagram possesses model elements that play the roles in the SPS.

Fig. 2(a) shows an SPS for a simplified Observer pattern [11]. In the SPS, there are two class roles, |*Subject* and |*Observer*. The |*Subject* role has a structural feature role |*state* which constrains that a class that plays the |*Subject* role must have an attribute playing the |*state* role. Similarly, the |*Observer* role has a behavioral feature role |*update(* that takes a parameter whose type is a subject class. Fig. 2(b) shows a class diagram instance that conforms to the SPS.

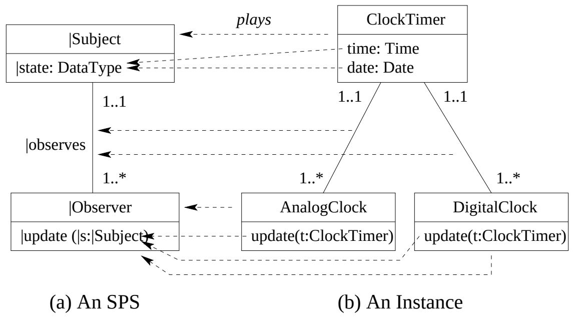


Figure 2: An SPS and a Class Diagram Instance

The class diagram has one class (*ClockTimer*) playing the |*Subject* role and two classes (*AnalogClock* and *DigitalClock*) playing the (|*Observer*) role. The class *ClockTimer* is able to play the |*Subject* role because it has attributes that can play the |*state* feature role in |*Subject*. Similarly, The *AnalogClock* and *DigitalClock* classes are able to play the |*Observer* role because they have an operation that can play the |*update* feature role in the |*Observer* role. The operations are able to play the |*update* role because they have a parameter whose type is *ClockTimer* which plays the |*Subject* parameter role in the |*update* role.

An IPS defines an interaction view of pattern participants in terms of *lifeline* and *message* roles whose bases are the *Lifeline* and *Message* metaclasses, respectively. A lifeline role characterizes lifelines that are instances of a classifier that plays a classifier role in an associated SPS. A sequence diagram is said to conform to an IPS if the relative sequence of the messages is consistent with the sequence of message roles that the messages are playing in the IPS. We use the UML 2.0 sequence diagram notation to describe IPSs for a richer set of constructs, including constructs for packaging (e.g., **loop**) interactions.

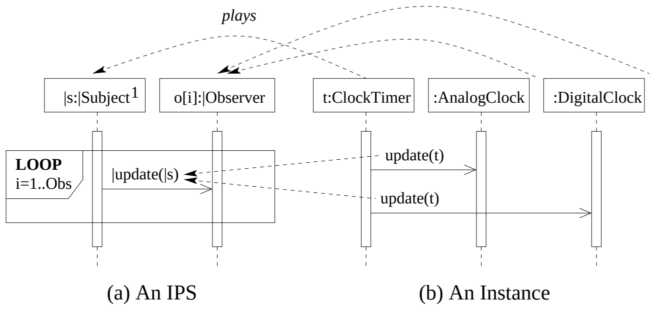


Figure 3: An IPS and a Sequence Diagram Instance

Fig. 3(a) shows an IPS for the Observer pattern. The IPS describes that an update message must be dispatched to every attached observer. Note that the **LOOP** fragment is a metamodel operator defined in the RBML [15], constraining the number of update messages to appear at the model level, and should not be confused with the *loop* operator in the UML [31]. *Obs* in the **LOOP** fragment is a function that returns the number of the attached observers.

3. ACCESS CONTROL MODELS AS DESIGN PATTERNS

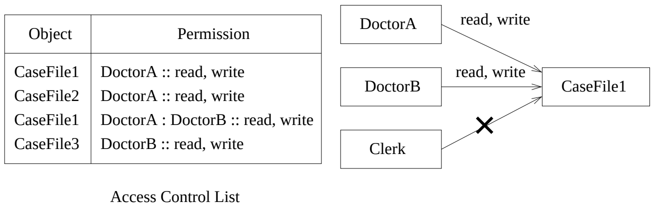
This section presents the DAC pattern based on Harrison *et al.*'s work[13], the MAC pattern based on the Bell-La Padula model for the MAC pattern [1], and the RBAC pattern based on the NIST proposal [9] and the book by Ferraiolo *et al.* [8].

3.1 Discretionary Access Control

The DAC pattern enforces access control based on user identities and the ownership of objects. The owner of an object may grant permission to another user to access the object, and the granted user may further delegate the permission to a third person.

Example

Consider an environment where access control is solely managed by the security administrator. A problem in such an environment is that it requires much effort for the administrator to maintain access control for every single user and to deal with daily-basis requests of permission changes. Another problem related to confidentiality is that the administrator may give unreasoned permission to a person who is not supposed to be authorized. For example, in a medical care system, access to patient information should be kept confidential and limited to only the doctor who handles the case, or other doctors who have permission given by the handling doctor. If the security administrator receives a fallacious request for permission given to a clerk, obviously it should not be allowed. In regard to availability, such an environment may cause a situation where no one can access information. For example, in the medical care system above, if *DoctorA* who handles *CaseFile1* has left for a vacation, without making a permission request for other doctors to access the file, no one can access the file in case of emergency. Fig. 4 illustrates these problems.



Object	Permission
CaseFile1	DoctorA :: read, write
CaseFile2	DoctorA :: read, write
CaseFile1	DoctorA : DoctorB :: read, write
CaseFile3	DoctorB :: read, write

Figure 4: A Motivating Example

Context

Development of access control systems that allow user-controlled administration of access rights to objects.

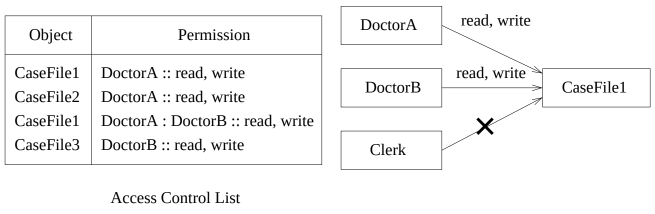
Problem

In an environment where access control is managed solely by administrators, unreasoned permission may be given to a person who should not be authorized, or no access may be allowed for any one. In such cases, it may be desired to leave access control decisions to the discretion of the owner of the respective resources. Use the DAC pattern:

- Where users own objects.
- When permission delegation is necessary.
- When a resolution for conflicting privileges is needed. For instance, a user may be allowed to access an object as a member of a group, but not allowed with individual permissions.
- When a security mechanism is needed in a heterogeneous environment for controlling access to different kinds of resources.
- Where multi-user relational database is used.

Solution

The DAC pattern can address the above problems by using the concept of "permission delegation" which allows a user of an object to give away permission to other users to access the object at her/his discretion without the intervention of the administrator. Using the DAC pattern, the burden on the administrator is shared with the users of objects who are capable of delegating permission. The DAC pattern mitigates the confidentiality problem above by granting permission directly to related people in the area. Also, the availability problem above can be addressed by delegating permission at the discretion of object users.



Object	Permission
CaseFile1	DoctorA :: read, write
CaseFile2	DoctorA :: read, write
CaseFile1	DoctorA : DoctorB :: read, write
CaseFile3	DoctorB :: read, write

Figure 5: DAC Solution

Fig. 5 shows an Access Control List (ACL) that implements the DAC pattern. In the ACL, a delegating user is represented in *user::rw* and a named user to which permission is delegated is represented in *user:namedUser:rw*. For example, *DoctorA* is a user of *CaseFile1*, and has read and write access to the file, and *DoctorB* is a named user who is granted access to *CaseFile1* by *DoctorA*, and has read and write access to the file.

An inconsistency between the permission given as an individual and the permission given as a member of a group to which the user belongs can be resolved by evaluating permission in an order. The evaluation stops either when all requested access rights have been granted by one or more permission entries, or when any one of the requested access rights has been denied by one of the permission entries.

Structure

Fig. 6 shows the solution structure of the DAC pattern.

- *User* represents a user or group who has access to an object, or a named user or group who are granted access to the object by the user or group. The owner or owning group of an object has full access to the object, and can grant or revoke permission to other users or groups at their discretion.
- *Object* represents any information resource (e.g., files, databases) to be protected in the system.
- *Operation* represents an action invoked by a user.

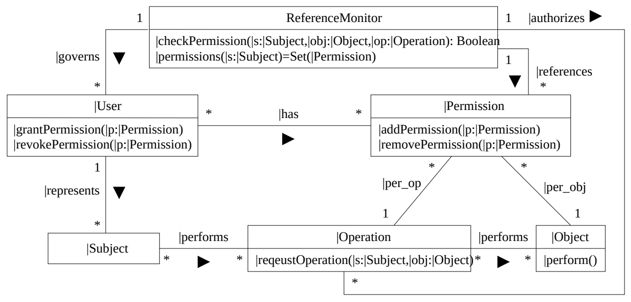

ReferenceMonitor
|checkPermission(|s:|Subject,|obj:|Object,|op:|Operation): Boolean
|permissions(|s:|Subject)=Set(|Permission)
|governs
|authorizes
|references
|User
|grantPermission(|p:|Permission)
|revokePermission(|p:|Permission)
|has
|Permission
|addPermission(|p:|Permission)
|removePermission(|p:|Permission)
|represents
|per_op
|per_obj
|Subject
|performs
|Operation
|requestOperation(|s:|Subject,|obj:|Object)
|performs
|Object
|perform()

Figure 6: DAC Solution Structure

- *Subject* represents a process acting on behalf of a user in a computer-based system. It could also be another computer system, a node or a set of attributes.
- *Permission* represents an authorization to carry out an action on the system. In general, Access Control Lists (ACLs) are used to describe DAC policies for its ease in reviewing. An ACL shows permissions in terms of objects, users and access rights.
- *ReferenceMonitor* checks permission for an access request based on DAC policies. If the user has permission to the object, the requested operation may be performed.

Dynamics

Fig. 7 shows the collaboration of the DAC pattern for requesting an operation. When an operation is requested for an object, the reference monitor intercepts the request and checks for permission based on DAC policies which is typically described in ACLs. If permission is found, the operation is performed on the object, otherwise, the request is denied.

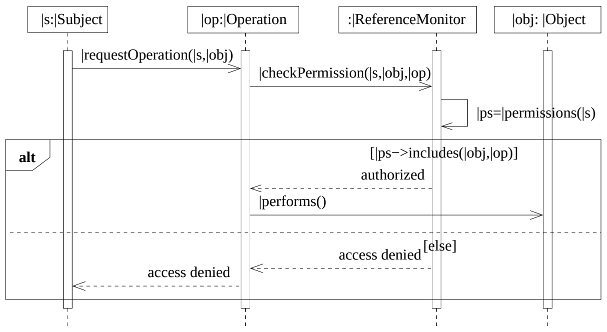


Figure 7: DAC Collaboration

Variants

Based on the underlying DAC concepts above, there are several variants [23] which are different by the degree of the strictness in owner's discretion. DAC variants can be categorized into strict DAC, liberal DAC and DAC with change of ownership. Strict DAC is the strictest form in which only the owner of an object can grant access to the object. Liberal DAC allows the owner to delegate access granting authority to other users. Liberal DAC can be further divided into one-level grant, two-level grant and multilevel-grant, depending on the level at which access granting authority can be passed on. DAC with change of ownership allows the owner to delegate ownership to other users.

Known Uses

Standard Oracle9i [19] uses the DAC pattern to mediate user access to data via database privileges such as SELECT, INSERT, UPDATE and DELETE. The TOE [32], a sensitive data protection product developed by The Common Criteria Evaluation and Validation Scheme (CCEVS), uses the DAC pattern to mediate access to cryptographic keys to prevent unauthorized access. Windows NT implements the DAC pattern to control generic access rights such as No Access, Read, Change, and Full Control for different types of groups (e.g., Everyone, Interactive, Network, Owner).

Consequences

The DAC pattern has the following advantages:

- Users can self manage access privileges.
- The burden of security administrators is significantly reduced, as resource users and administrators jointly manage permission.
- Per-user granularity for individual access decisions as well as coarse-grained access for groups are supported.
- It is easy to change privileges.
- Supporting new privileges is easy.

The DAC pattern has the following disadvantages:

- It is not appropriate for multilayered systems where information flow is restricted.
- There is no mechanism for restricting rights other than revoking the privilege.
- It becomes quickly complicated and difficult to maintain access rights as the number of users and resources increases.
- It is difficult to judge the "reasonable rights" for a user or group.
- Inconsistencies in policies are possible due to individual delegation of permission.
- Access may be given to users that are unknown to the owner of the object. This is possible since the user granted authority by the owner can give away access to other users.

See Also

The Authorization pattern [7] which addresses accessing objects by subjects. The Authorization pattern has the concept of delegation as in the DAC pattern. However, unlike the DAC pattern, the access request may need not specify a particular object in the rule. It may be implied by the existing objects being protected.

3.2 Mandatory Access Control

The MAC pattern governs access based on the security level of subjects (e.g., users) and objects (e.g., data). Access to an object is granted only if the security levels of the subject and the object satisfy certain constraints. The MAC pattern is also known as multilevel security model and lattice-based access control.

Example

The MAC pattern addresses the following two problems in the DAC pattern. First, there is nothing that prevents a user who is granted read access to a file by the owner of the file from copying the content of the file and granting read access to other users. For example, consider Fig. 8. Let's suppose *John* wants to access *File1* which he does not have permission to access. Of course, a DAC system will not allow him to access the file since he has no permission. However, if there is a third person who has read access to *File1* granted by (*Jane*) who is the owner of *File1*, and the person grants *John* read access to *File1*, then *John* can read *File1* without *Jane* being aware of it. Secondly, a user who has write access to a file may write a Trojan horse program into the file to copy the contents of the file. A Trojan horse program disguises as if it is a utility program while copying file contents. In Fig. 8, the Trojan horse program copies the contents of *File1* into *File2* which *John* can access.

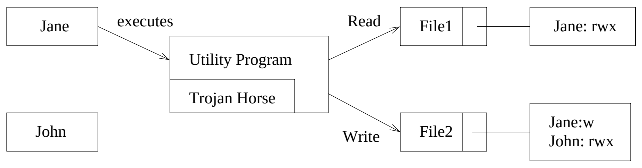


Figure 8: Trojan Horse Problem in the DAC Pattern

Context

Development of access control systems that handle classified objects and need to limit users' actions according to a hierarchy of classifications.

Problem

The MAC pattern can solve the above problems with the DAC pattern in a multi-layered environment (e.g., military and government systems) by assigning security levels to users and objects. In this sense, the solution of the DAC pattern can be considered as a problem of the MAC pattern. That is, if a DAC system is deployed in a multi-layered environment, the MAC pattern can be applied to improve the confidentiality of the system. Use the MAC pattern:

- Where the environment is multi-layered. For example, in the military domain, users and files are classified into distinct levels of hierarchy (e.g., Unclassified, Public, Secret, Top Secret), and user access to files is restricted based on the classification.
- When security policies need to be defined centrally. Access control decisions are to be imposed by a mediator (e.g., security administrator), and users should not be able to manipulate them.

Solution

The MAC pattern solves the above issues in a classified environment by assigning security levels to users and objects. In the MAC pattern, a user can read a file, but cannot write to the file if the user's security level is higher than or equal to the file's security level. A user can write to a file, but cannot read the file if the user's security level is lower than or equals to the file's security level. For example, consider the military domain where documents are classified and categorized as shown in Fig. 9

Classification	Category
UNCLASSIFIED	U.S.
CONFIDENTIAL	U.S.
SECRET	U.S.
TOP SECRET	U.S.
UNCLASSIFIED	Allies
CONFIDENTIAL	Allies
SECRET	Allies
TOP SECRET	Allies

User	Classification	Category
John	SECRET	US
Jane	TOP SECRET	US
Smith	UNCLASSIFIED	Allies
Bill	UNCLASSIFIED	US

File	Classification	Category
File1	SECRET	US

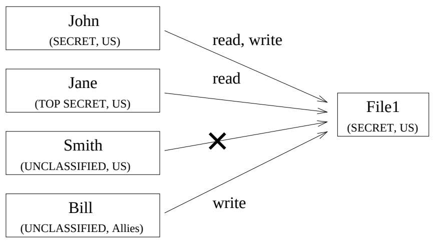


Figure 9: A MAC Example

In the diagram, *John* has read and write access to *File1* since his classification and category are same as that of *File1*. However, *Jane* can only read *File1*, but cannot write to the file because the classification of *File1* is lower than that of *Jane*. *Smith* cannot read and write to *File1* because his category is different from the category of *File1*. *Bill* can write to *File1*, but cannot read the file because the classification of the file dominates his classification.

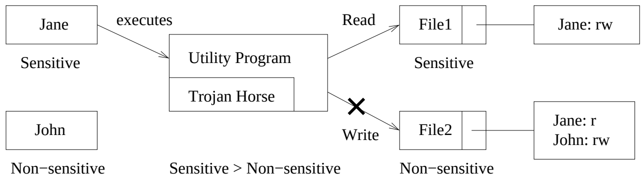


Figure 10: Trojan Horse Solution in the MAC Pattern

Using the MAC pattern, the Trojan horse problem in Fig. 8 can be resolved as shown in Fig. 10. In Fig. 10, the Trojan horse program running on behalf of *Jane* can read *File1* since the security level of *Jane* is equal to that of *File1*. However, it cannot write to *File2* because *Jane*'s security level is lower than the security level of *Fil2*. Thus, *John* is not able to access *File1*.

Structure

Fig. 11 shows the solution structure of the MAC pattern [17].

- *User* represents a user or a group of users who interacts with the system. A user is assigned a hierarchical security level (e.g., SECRET, CONFIDENTIAL) and non-hierarchical category (e.g., U.S., Allies) to which the user belongs. A user may have multiple login IDs

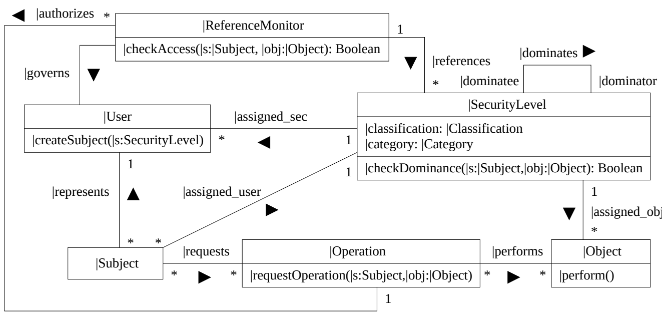


Figure 11: MAC Solution Structure

which can be activated simultaneously. A user also may create and delete one or more subjects.

- *Subject* represents a computer process that acts on behalf of a user to request an operation on an object. For instance, an ATM machine being used by a user can be viewed as a subject. A subject may be given the same security level as the user or any level below the user's security level.
- *Object* represents any information resource (e.g., files, databases) in the system that can be accessed by the user. Similar to users, an object is assigned a hierarchical security level and a non-hierarchical category to which the object belongs.
- *Operation* is an action being performed on an object invoked by a subject.
- *SecurityLevel* represents a sensitivity assigned to users (subjects) and objects. A security level consists of a classification and a category. While classifications are hierarchical, categories are non-hierarchical.
- *ReferenceMonitor* checks accessibility based on the following constraints.
  - Simple security property - A subject S is allowed read access to an object O only if $L(S) \geq L(O)$.
  - Star property - A subject S is allowed write access to an object O only if $L(S) \leq L(O)$.

  Access is allowed when both the constraints are satisfied. Access is checked only if the user is in the same category as that of the object. With the categories matched, the accessibility of the user for the object is determined by the dominance relations of classifications in the above constraints.

Dynamics

Fig. 12 shows the collaboration for requesting an operation. The diagram describes that a subject requests an operation on an object, and the request is intercepted by the reference monitor to check authorization by enforcing the simple security property and star property. The request is performed on the object if it is authorized, otherwise it is denied.

Variants

The Biba Integrity model [2] can be viewed as a variant of the MAC pattern, emphasizing on integrity rather than confidentiality. Similar to the MAC pattern, Biba model has *Simple Integrity property* and *Integrity Star property* which are defined as follows:

- Simple integrity property - A subject S is allowed read access to an object O only if $L(O) \geq L(S)$.
- Integrity star property - A subject S is allowed write access to an object O only if $L(O) \leq L(S)$.

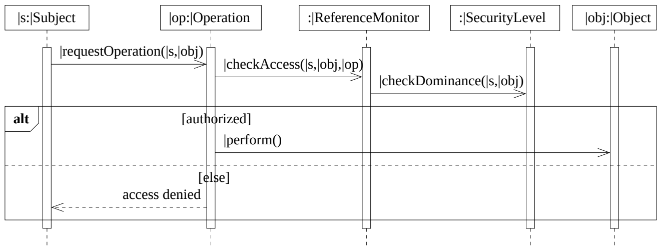


Figure 12: MAC Colloboration

Known Uses

Security-Enhanced Linux (SELinux) kernel [27] developed by a collaboration of NSA, MITRE Corporation, NAI labs and Secure Computing Corporation (SCC) enforces the MAC pattern to implement a flexible and fine-grained MAC architecture called Flask which operates independently of the traditional Linux access control mechanisms. TrustedBSD [33] developed by the FreeBSD Foundation provides a set of trusted operating system extensions to the FreeBSD operating system which is an advanced operating system for x86, amd64 and IA-64 compatible architectures. TrustedBSD contains modules that implement MLS (Multi-Level Security) and fixed-label Biba integrity policies which is a variant of the MAC pattern. GeSWall (General Systems Wall) [12] is the Windows security project developed by GentleSecurity. GeSWall implements the MAC pattern to provide OS integrity and data confidentiality transparent and invisible to user.

Consequences

The MAC pattern has the following advantages:

- MAC systems are secure to Trojan horse attacks.
- The assignment of a classification and category to users and objects is centralized by a mediator.
- The MAC pattern facilitates enforcing access control policies based on security levels.

The MAC pattern has the following disadvantages:

- Introducing a new object or user requires a careful assignment of a classification and category.
- The mediator who assigns classifications to users and objects should be a trusted person.

See Also

The Biba's Integrity model [2] which addresses integrity issues where access is determined by the integrity levels of subjects and objects, rather than confidentiality. The Chinese Wall model [3] which is similar to the MAC pattern in that it also defines read and write constraints where the write constraint takes into account the Trojan horse problem. However, unlike the MAC pattern, the Chinese Wall model does not distinguish between users and subjects. Subjects include both users and processes acting on behalf of the user.

3.3 Role-Based Access Control

The RBAC pattern enforces access control based on roles. A role is given a set of permissions, and the users assigned to the role acquires the permissions given to the role. Since the RBAC pattern is based on roles which are in general fewer than the number of users, it is useful for managing a large number of users.

Example

In a small organization where the application of an information system is narrow and the number of users and objects are low, user-based access control (e.g., the DAC pattern) where users are directly mapped to permissions could be used. However, in a large organization, user-based access control becomes infeasible due to too many mapping instances. For example, suppose that a person is hired as a secretary and given a certain set of privileges. In user-based access control, *n* number of mapping instances are required, where *n* is the number of privileges to be given. If the person is later moved to another position, all the privileges given as a secretary have to be revoked, requiring the same amount of effort as assigning privileges for the secretary position. Also, a different set of privileges should be assigned for the new position, requiring another huge amount of effort.

Context

Development of access control systems that handle a large number of users and objects and are expected to have frequent changes of access rights.

Problem

The DAC pattern is not suitable for managing a large number of users and objects. The DAC rose from small and autonomous environments, and as such the DAC pattern is usually used in academia and small organizations. Similarly, the MAC pattern is limited to environments where users and information are classified, for example the government and military domains. As far as security is concerned, the MAC pattern addresses the Trojan horse problem in the DAC pattern as shown in Subsection 3.2. However, the MAC pattern still allows some security breaches known as "Covert Channels" (e.g., storage channels, timing channels) which reveals certain information of the system via a Trojan horse program. For example, a Trojan horse program may transmit information such as when the program runs or waits or the usability of shared resources (e.g., by creating and deleting bogus print jobs). Use the RBAC pattern:

- Where control of data and application is restricted to the enterprise (i.e., users do not "own" data).
- Where there are a large number of users and data objects to control (e.g., e-commerce applications in cross-enterprise distributed networks).
- When a limited number of security administrators are available.
- When the organization structure is stable (i.e., no frequent changes of job definitions).
- When there is frequent change of job responsibility or high job turnover.

Solution

The RBAC pattern overcomes the above problems by using roles (e.g., Secretary, Manager) which are an abstraction of users. Instead of directly mapping users to permissions, the RBAC pattern maps roles to permissions and assigns users to the roles for which the users are authorized. The users assigned to a role acquire the permissions given to the role. Fig. 13 shows an implementation of the RBAC pattern in the hospital domain. In the figure, the objects *Prescription* and *CaseFile* are mapped to roles of *Physician*, *Nurse* and *Patient* with permissions to the objects. For example, the *Physician* role is given read and write permissions to the *Prescription* and *CaseFile* objects. *John* and *Joe* are assigned to the *Physician* role, and they acquire the read and write permissions given to the role.

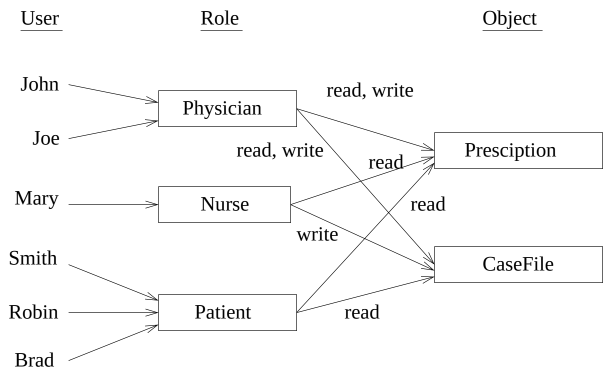


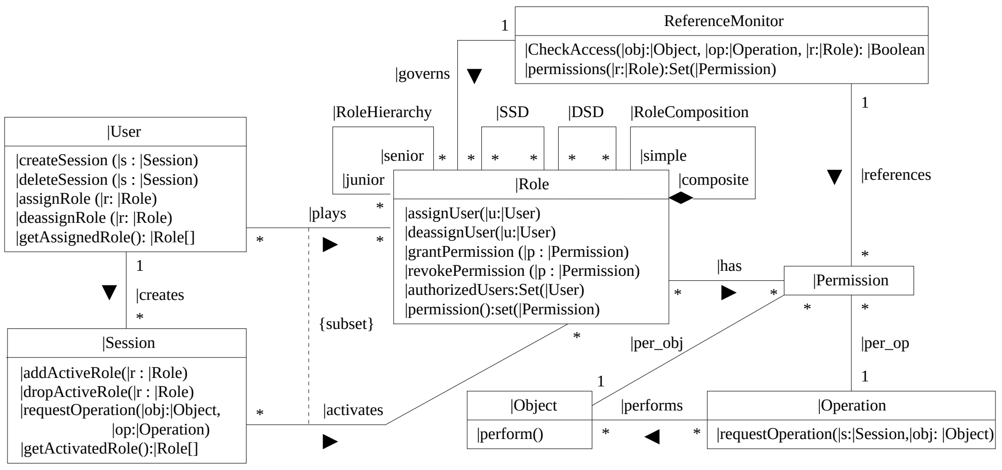
Figure 13: RBAC Example

Since the RBAC pattern is based on relatively static entities (roles and permissions), access control is simple and efficient. For example, in Fig. 13 if *John* moves to another position such as surgeon, *John* can be simply deassigned from the *Physician* role to revoke his privileges as a physician and assigned to the surgeon role to acquire privileges as a surgeon.

Structure

The RBAC pattern involves the concepts of *Role*, *User*, *Session*, *Object*, *Permission* and *Operation* as shown in Fig. 14.

- *Role* represents a job function with certain authority and responsibility in an organization. A role can be represented as a relation of a set of users and a set of permissions. A user assigned to a role acquires the permission set defined in the role relation. Roles may have overlapping responsibilities and rights.
  Roles may be structured in a hierarchy to reflect an organization's lines of authority and responsibility. A role hierarchy defines an inheritance relation among roles in terms of permissions and user assignments. That is, a role *r1* inherits another role *r2* if only if 1) the permissions of *r2* is a subset of the permissions of *r1*, and 2) the users of *r1* is a subset of the users of *r2*.
  Two roles may have a conflict of interests that prevents a user from being assigned to both the roles. That is, a user who has a membership in one role cannot be a member of the other conflicting role. This is called *Static Separation of Duty* (SSD). SSD constraints are enforced during user assignment.
  In a role hierarchy, two junior roles in an SSD relation



ReferenceMonitor
|CheckAccess(|obj:|Object, |op:|Operation, |r:|Role): |Boolean
|permissions(|r:|Role):Set(|Permission)
|governs
|User
|createSession (|s : |Session)
|deleteSession (|s : |Session)
|assignRole (|r: |Role)
|deassignRole (|r: |Role)
|getAssignedRole(): |Role[]
|creates
|Session
|addActiveRole(|r : |Role)
|dropActiveRole(|r : |Role)
|requestOperation(|obj:|Object, |op:|Operation)
|getActivatedRole():|Role[]
|RoleHierarchy
|senior
|junior
|SSD
|DSD
|RoleComposition
|simple
|composite
|Role
|assignUser(|u:|User)
|deassignUser(|u:|User)
|grantPermission (|p : |Permission)
|revokePermission (|p : |Permission)
|authorizedUsers:Set(|User)
|permission():set(|Permission)
|plays
{subset}
|activates
|references
|has
|Permission
|per_obj
|per_op
|Object
|perform()
|performs
|Operation
|requestOperation(|s:|Session,|obj: |Object)

Figure 14: Solution Structure

may have the same senior role. This causes a security breach because the senior inherits the conflicting privileges from the junior roles. Thus, it should be prevented. This is called Hierarchical SSD.
A user may be assigned to two roles, but the roles may not be activated simultaneously within the same user session. This is called *Dynamic Separation of Duty* (DSD) which is enforced during role activation within a session. If one role in a DSD relation is activated, the other role in the relation cannot be activated in the same session.

- *User* is a person who interacts with a computer system. A user may have multiple login IDs which can be activated simultaneously. A user can create and delete a session.
- *Session* represents an instance of a user's dialog with a system. A session can be represented as a mapping of a user and a set of activated roles. A session can activate and deactivate a role, and a user may have multiple sessions running simultaneously.
- *Object* represents any information resource (e.g., files, databases) being protected in the system.
- *Operation* is an action to be performed on an object, invoked within a session. Examples of operations are read, write and execute in a file system and insert, delete, append and update in a database management system.
- *Permission* represents an authorization to perform an operation on an object or on multiple objects. A permission is composed of an operation and an object on which the operation is performed. Access is denied if a permission is not found for the requested operation and the target object.
- *ReferenceMonitor* checks accessibility by enforcing SSD, DSD and role hierarchy constraints.

Dynamics

Fig. 15 shows the collaboration for checking access. The diagram describes that an operation request is invoked within a session, and the request is intercepted by the reference monitor to check accessibility. The request is checked against the permission set of each role activated in the session which is specified by the *loop* fragment. If a permission is found, the request is allowed, otherwise, it is denied.

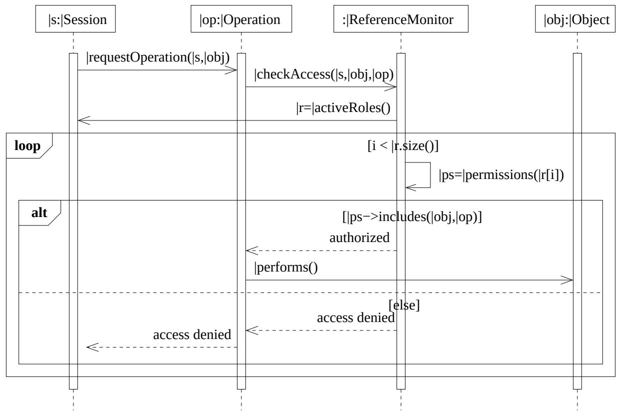


Figure 15: Requesting Operation

Variants

There are four variants (RBAC0, RBAC1, RBAC2, RBAC3) of the RBAC pattern which are in a build-up relationship. RBAC0 describes the fundamental concepts of RBAC including *Role*, *User*, *Operation*, *Object* and *Permission*. RBAC1 adds role hierarchies to RBAC0, and RBAC2 adds SoD (Separation of Duty) to RBAC0. RBAC3 is a combination of RBAC0, RBAC1 and RBAC2. The RBAC pattern presented in this paper is RBAC3. The RBAC pattern may be used with the DAC pattern [23] and MAC pattern [20, 22] to complement each other.

Known Uses

The Sun ONE Identity Server [28] uses the RBAC pattern to map business functions to a logical group of users using roles. Sun's J2EE [28] uses the RBAC pattern in the authorization service (e.g., password management systems in UNIX and Windows) for E-Commerce applications [29]. Solaris 8 uses the RBAC pattern to restrict access to tools and utilities.

IBM uses the RBAC pattern for security within WebSphere Portal [34] where users are categorized into roles of guests, users, administrators and super users. In Oracle applications, roles are used to determine what data and functions within an application a user has access to [18].

Consequences

The RBAC pattern has the following advantages:

- It reduces the complexity of access control for a large number of users and objects. This also facilitates updating access rights of users.
- Changes of organization policies about job functions can be dynamically reflected without interrupting user's work through changes of role definitions.
- A user can activate multiple sessions at a time, and a session can activate multiple roles assigned to the user, which provides functional flexibility,
- It supports the least privilege principle which describes that only the necessary privileges should be given to the user to perform their duties.
- Granting and revoking permissions are easy.

The RBAC pattern has the disadvantage:

- The additional concepts (e.g., roles, sessions) and their related constraints (e.g., SSD, DSD) add complexity to implementation.

See Also

The Abstract Session pattern [21] which is similar to the RBAC pattern, but gives more focus on network sessions. In general, both the patterns are used in a networking environment. Using the Abstract Session pattern, an object can store per-client state without sacrificing type-safety or efficiency through session creation. The Thread Specific Storage pattern [26] which allows multiple threads to use one logically global access point to retrieve thread-specific data without incurring locking overhead. This is similar to the RBAC pattern in the sense that multiple users can play a single role to perform role-specific operations.

4. CONCLUSION

We have presented DAC, MAC and RBAC as design patterns using the POSA template. We use the RBML to capture the variations of the structure and behavior of the patterns. We have attempted to provide more details on the problem domain of the patterns to help the developer choose a suitable pattern for a given problem. Also, the problem descriptions can be used as a basis for formalizing the problem domain for systematic checking of pattern applicability.

Acknowledgements

The authors would like to thank Eduardo Fernandez who is a shepherd of our paper for his helpful comments and suggestions. This work is supported in part by the National Science Foundation under Grant No. CCF-0523101.

5. REFERENCES

[1] D. E. Bell and L. J. LaPadula. Secure Computer System: Unified Exposition and MULTICS Interpretation. Technical Report MTR-2997, MITRE Corporation, Bedford, MA, July 1975.

[2] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, Bedford, MA. Air Force Electronic Systems Division, 1977.

[3] D. F. Brewer and J. Nash. The Chinese Wall Security Policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 206–258, Oakland, California, 1989.

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

[5] F. Chen and R. Sandhu. Constraints for Role-Based Access Control. In *Proceedings of the 1st ACM Workshop on Role-Based Access Control*, Gaithersburg, MD, 1995.

[6] T. Doan, S. Demurjian, T.C. Ting, and A. Ketterl. MAC and UML for Secure Software Design. In *Proceedings of 2nd ACM Workshop on Formal Methods in Security Engineering: From Specifications to Code*, Washington D.C., 2004.

[7] E. B. Fernandez and Pan R. A Pattern Language for Security Models. In *Proceedings of the 8th Conference on Pattern Language of Programs (PloP)*, Monticello, IL, 2001.

[8] D. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-Based Access Control*. Artech House, 2003.

[9] 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 Systems Security*, 4(3), 2001.

[10] R. France, D. Kim, S. Ghosh, and E. Song. A UML-Based Pattern Specification Technique. *IEEE Transactions on Software Engineering*, 30(3):193–206, 2004.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[12] General System Wall. http://www.securesize.com/GeSWall/.

[13] M.H. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in Operating Systems. *Communications of the ACM*, 19(8):461–471, 1976.

[14] D. Kim. *A Meta-Modeling Approach to Specifying Patterns*. PhD thesis, Colorado State University, Fort Collins, CO, 2004.

[15] D. Kim. The Role-Based Metamodeling Language for Specifying Design Patterns. In Toufik Taibi, editor, *Design Pattern Formalization Techniques*, pages 183–205. Idea Group Inc., 2007.

[16] D. Kim, R. France, S. Ghosh, and E. Song. A Role-Based Metamodeling Approach to Specifying Design Patterns. In *Proceedings of the 27th IEEE Annual International Computer Software and Applications Conference*, pages 452–457, Dallas, Texas, 2003.

[17] D. Kim and P. Gokhale. A Pattern-Based Technique for Developing UML Models of Access Control Systems. In *Proceedings of the 30th Annual International Computer Software and Applications Conference*, pages 317–324, Chigaco, IL, 2006. IEEE Computer Society Press.

[18] L. Notargiacomo. Role-Based Access Control In ORACLE7 And Trusted ORACLE7. In *Proceedings of the 1st ACM Workshop on Role-Based Access Control*, page 17, Gaithersburg, MD, 1995.

[19] Oracle9i. http://www.oracle.com/technology/-products/oracle9i/datasheets/ols/OLS9iR2_ds.html.

[20] S. L. Osborn, R. Sandhu, and Q. Munawer. Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies. *ACM Transactions on Information and System Security*, 3(2):85–106, 2000.

[21] N. Pryce. Abstract Session: an object structural pattern. In L. Rising, editor, *Design Patterns inCommunications Software*, pages 191–208. Cambridge University Press, New York, USA, 2001.

[22] I. Ray, N. Li, D. Kim, and R. France. Using Parameterized UML to Specify and Compose Access Control Models. In *Proceedings of the 6th IFIP TC-11 WG 11.5 Working Conference on Integrity and Internal Control in Information Systems (IICIS)*, Lausanne, Switzerland, 2003.

[23] R. Sandhu and Q. Munawer. How To Do Discretionary Access Control Using Roles. In *Proceedings of the 3rd ACM Workshop on Role-Based Access Control (RBAC-98)*, Fairfax, VA, 1998. ACM Press.

[24] R. Sandhu and P. Samarati. Access Control: Principles and Practice. *IEEE Communications*, 32(9):40–48, September 1994.

[25] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[26] D. C. Schmidt, T.H. Harrison, and N. Pryce. Thread-Specific Storage - An Object Behavioral Pattern for Accessing per-Thread State Efficiently. *In the C++ Report*, 9(10), 1997.

[27] SELinux. http://www.nsa.gov/selinux/.

[28] Sun One Identity Server. http://sunflash.sun.com/articles/62/4/iplanet/9662.

[29] M. M. Swift, A. Hopkins, P. Brundrett, C. Van Dyke, P. Garg, S. Chan, M. Goertzel, and G. Jensenworth. Improving the granularity of access control for Windows 2000. *ACM Trans. on Information and System Security*, 5(4):398–437, November 2002.

[30] The Object Management Group (OMG). Unified Modeling Language. Version 1.5, OMG, http://www.omg.org, March 2003.

[31] The Object Management Group (OMG). Unified Modeling Language: Superstructure. Version 2.1.2 formal/07-11-02, OMG, http://www.omg.org, November 2007.

[32] TOE. http://niap.nist.gov/cc/-scheme/st/ST_VID4048.html.

[33] TrustedBSD Project. http://www.trustedbsd.org/.

[34] WebSphere Portal for Multiplatforms. http://www-306.ibm.com/software/genservers/portal/.