A PATTERN LANGUAGE FOR DESIGNING AND IMPLEMENTING ROLE-BASED ACCESS CONTROL

Saluka R. Kodituwakku[1], Peter Bertok[1], and Liping Zhao[2]
[1]Department of Computer Science
RMIT University, Australia
{sakoditu,pbertok}@cs.rmit.edu.au

[2]Department of Computation, UMIST, UK
liping@co.umist.ac.uk

Abstract

This paper presents a collection of object-oriented design patterns structured to form a pattern language for Role Based Access Control (RBAC). A hypothetic banking application is used to describe how these patterns support the design and implementation of RBAC models.

1. Introduction

Access control is one of the core issues in system security, and several access control models, such as discretionary access control (DAC), mandatory access control (MAC) [8] and role based access control [6, 7, 13] have been developed. Role-based access control (RBAC) is based on the application area's organizational characteristics such as structure of data models and security policies.

RBAC is a family of reference models [13] in which privileges are associated with roles and roles are associated with users. Some variations of RBAC [6, 7] include the capabilities of establishing relationships between privileges and roles, and between users and roles.

Using object-oriented approach, roles can be represented as objects and privileges as access rights on methods. When roles have overlapping responsibilities, inheritance can be used to design a hierarchy of roles in such a way that any role can execute all methods available to its child classes.

This paper shows that the Object-Oriented paradigm is a promising approach to role-based access control (RBAC). It presents a model that can be used by administrators to administer and regulate user access to information objects in a manner that is consistent with organizational security policies. The paper is organised as follows. Section 2 summarizes access control models. Section 3 describes a set of patterns for designing and implementing RBAC. Section 4 discusses the results and section 5 concludes the paper.

2. Access Control Models

Access control is concerned with restricting the activity of legitimate users. It relies on and coexists with other security services in a computer system. We have to make a clear distinction between authentication and authorization (access control): authentication is used for correctly establishing user identity, access control assumes that the user has already been successfully identified by the authentication service.

To ensure that authorized users do not misuse their privileges, access control needs to be combined with auditing, when a posterior analysis of requests and activities of users in the system is performed.

Policies or high level guidelines that determine how access is controlled and access decisions are made, have to be in harmony with software and hardware mechanisms, that are used to implement policies.

2.1 The Access Matrix

The basic abstractions in access control are objects and subjects. Resources are represented by data stored in objects, while activities in the system are initiated by entities known as subjects. Subjects are typically users or programs executing on behalf of users. A user may sign on to the system as different subjects on different occasions, depending on the privileges the user needs.

Operations on objects are permitted or denied according to authorizations expressed in terms of access rights or access modes.

The access matrix is a conceptual model that specifies the access rights of each subject for each object. In the matrix, a row describes a subject and a column describes an object. Hence a matrix element specifies the access rights of

a particular subject on a particular object. The major task of access control is to ensure that only operations authorized by the access matrix will get executed.

2.2 Access Control Policies

2.2.1 Discretionary Access Control (DAC) and Mandatory Access Control (MAC) Policies

DAC protection policies govern the access of users to the information on the basis of the user's identity and authorizations that specify, for each user or group of users and each object in the system, access modes the user is allowed on individual objects. Each request is validated against the specified authorizations. If the user is authorized to access the object in the specified mode, the access is granted, otherwise it is denied. DAC policies provide great flexibility that is not always desirable, e.g. policies may allow users to grant or revoke access to objects under their control without input from a security administrator.

MAC policies do not allow users to grant or revoke access rights to objects, a security administrator is responsible for that. MAC policies prevent the information flow from high level objects to low-level objects, and they are significantly less flexible than DAC policies.

2.2.2 Role Based Access Control Policies

Role-based access control policies govern access to information on the basis of activities of users [6, 7, 13]. First roles are identified in the system, then privileges to access objects are specified for roles. A user acting in a role is allowed to execute all accesses for which the role is authorized. Some RBAC implementations allow a user to act in multiple roles at the same time, others restrict the user to one role at a time.

Users, Roles, Subjects, Privileges and Objects

In a RBAC environment a *user* is a person working with the system, a *role* is a collection of job functions (job title) that a user or set of users can perform within the context of an organization. A subject represents an active user process and each subject is a mapping of a user to one or possibly many roles. A unique user identifier determines the user associated with the subject. The application environment defines the association between roles and privileges. The following two figures show the associations between user, subject, role, object and privileges.

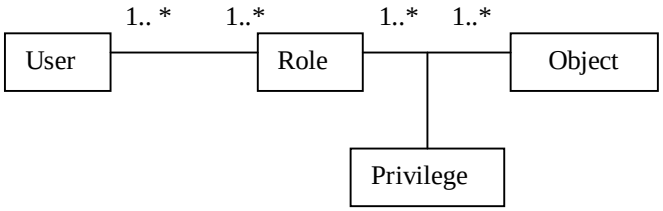

Figure 1. Associations between user, role, object and privilege

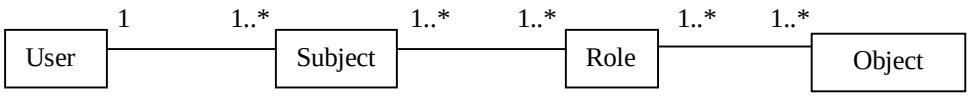

Figure 2. Association between user, subject, role and object

The mapping among users, subjects and roles are described by the following functions:

2. *Subject-user (s: subject) = { the user associated with subject "s"}*
3. *Authorized-role (s: subject) = { the roles associated with subject "s"}*
4. *Role-members (r: role) = { the users authorized for role "r'}*
5. *User-authorized-roles (u: user) = { the roles associated with user "u"}*

Let R = *Authorized-role (s*) and *u = Subject-user (s).* Then the user "*u*" must also be associated with the set of roles *"R"*.

Objects and Operations

The set of objects covered by the RBAC system includes all of the objects accessible by RBAC operations. An operation represents a particular unit of control that can be referenced by an individual role that is subject to constraints within the RBAC system. An operation can be used to capture security relevant details or constraints that can not be determined by a simple mode of access. These details can be in the method or in the granularity of access. The type of operation and the objects that RBAC manages are dependent on the type of the system in which it is implemented. For example, within a database management system, operations might include insert, delete and update. The association between objects and operations are shown in figure 3.

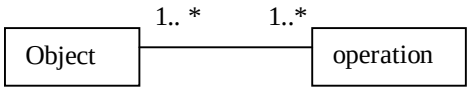

Figure 3. Association between objects and operations

The relationships between roles and operations, and objects and operations are described by the following functions:

1. *Role-operation (r: roles) = { the operations that are associated with role "r"}*
2. *Operation-objects (op: operation) = {the authorized objects to which the operation "op" can be applied}*

The role-based approach has the following advantages:

- **Authorization management**: Role-based policies benefit from a logical independence in specifying user authorizations by breaking this task into two parts, user-role assignment and role-privilege assignment.
- **Role Hierarchies**: In many applications there is a natural hierarchy of roles, based on generalization and specialization. Role hierarchies can be used to support roles with overlapping responsibilities and privileges, as users having different roles may need to perform the same operations. A role hierarchy defines roles that have unique attributes and each role may implicitly include the operations and, constraints that are associated with other roles in the higher levels of the hierarchy. The following diagram shows some possible roles, job titles and classes in a branch office of a bank.

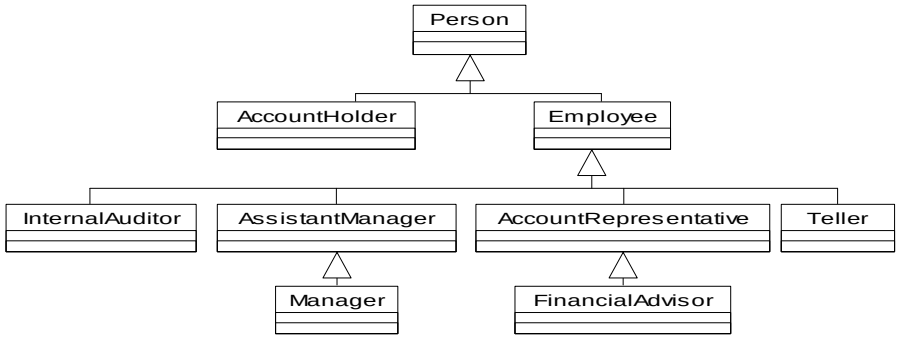

Figure 4. Roles, job titles and classes in a branch office of a bank

- **Principle of least privileges:**This constraint says that the minimum set of privileges required to invoke an operation should be granted to a user requesting that operation.
- **Mutually exclusive roles:** Two roles are said to be mutually exclusive, if a user is not allowed to act in both roles. A user can be assigned a role only if that role is not mutually exclusive with any of the roles the user already has. This is known as static separation of duty. This can be formally described as follows:

$$\forall u: user, r_i, r_j: roles: i \neq j: u \in role\text{-}members(r_i) \wedge u \in role\text{-}members(r_j) \Rightarrow r_i \notin mutually\text{-}exclusive(r_j).$$

- **Cardinality constraint:**Some roles can only be taken by a limited number of users at any point in time. This is known as cardinality constraint. Cardinality constraint is formally described by the following equations:

1. *Membership-limit (r: roles) = the membership limit (≥ 0) for role "r".*
2. *Number-of-members (r: roles) = N (≥ 0) the number of existing members in role "r".*
3. *r: roles: Membership-limit (r) $\geq$ Number-of-members (r).*

2.3 Administration of Authorization

Administrative policies determine who is authorized to modify the allowed accesses. In DAC, security administrator grant and revoke access rights to users. Users are allowed to grant or revoke access rights for other users to object they created. In MAC, access authorizations are entirely determined on the basis of the security classification of subjects and objects. The administrator assigns security levels to users. Security levels of objects are determined by the system on the basis of the levels of the users creating them. Administrator is the only one who can change the security levels of subjects or objects. In RBAC, roles can be used to manage and control the administrative mechanism. However, these roles, administrative roles, are distinct from common roles.

Delegation of administrative responsibilities is very important in which central administration is infeasible or an existing access control system is inefficient. Some systems allow administrative responsibilities for a particular set of objects to be delegated by the central administrator to other administrators. For, example, in distributed systems, administrative responsibilities to administer object in a local domain can be granted to the local administrator.

2.4 The example application

To illustrate the use of RABC, a banking application was chosen. The roles modelled are Manager, Financial Adviser, Account Representative, Internal Auditor, Teller, Account Holder and Person as illustrated in figure 4.

The following *Use Cases* describe the required interactions of users with the application:

- Create a new account
- Delete an existing account
- Read account information
- Write account data (deposit and withdrawal)
- Modify (make corrections to) account data
- Approve or reject customer loans

In the application the following roles and constraints apply.

- Account holders can read their own account records.
- Account holders can withdraw funds from their own accounts only but are permitted to deposit funds to any account.
- All employees are permitted to read account information.
- Employees can have accounts in the bank but may not access their personal accounts while dealing with other accounts.
- Account representative and internal auditor are mutually exclusive roles.
- The same person may act as Account representative, Teller or Account holder. However, he is not allowed to play these roles at the same time.
- A Teller role is permitted to read and write account data.
- An Account representative is permitted to create, delete, and modify accounts and perform any corrections
- An assistant manager is allowed to approve customer loans not exceeding 50,000 dollars.

3. Pattern Language

3.1 Descriptive Form

The description form used here is derived from GOF form and all Patterns are described with the following components. Name, context, problem description, solution, benefits and drawbacks, resultant context, i.e. situation after application of the pattern, related patterns and Known Uses is also documented together with implementation examples in the form of code fragments in C++.

3.2 The patterns

Pattern	Purpose
1. Role-Based-Access	Introduces roles to access protected information objects on behalf of users and introduces separate roles to administer users and roles.

2. Administrator	Handles user-role assignment and delegates the administrative responsibilities.
3. Access-Controlled-by-Roles	Defines roles to control user access to information.
4. Object-Collection	Creates complex role operations as a collection of simple objects.
5. Proxy-Role	Implement role classes based on privileges of the corresponding role.
6. Role-Selection-Strategy	Forms role hierarchies and implements the different access privileges of different roles.

Role Based Access

Context
Direct access to information objects is technically possible but may lead to misuse of organizational resources. You want to control user access to information objects so users are permitted to invoke only operations explicitly corresponding to their responsibilities.

Problem
RBAC needs a balance between the following forces.

- Users in an organization have different responsibilities that vary with job titles.
- Users can issue requests that are not part of their responsibilities. However, these requests have to be denied by RBAC.
- Users may have different or overlapping privileges. Overlapping responsibilities can form hierarchies. Users are allowed to invoke all operations available to users above them in the hierarchy.
- Some restrictions apply to roles, e.g. employees have restricted access to their own account records.
- Administration of user-role assignment and role-privileges assignment must be easy and flexible.

Solution
- Identify job titles and user responsibilities.
- Introduce a set of roles and a set of privileges for each job title.
- Introduce a separate set of roles and privileges to administer user-role assignment and role-privileges assignment. If centralised administration can not be implemented, divide the administrator responsibilities and introduce local administrator roles and privileges and a chief administrator role and privileges. The chief administrator is responsible for managing local administrator roles.

Consequences
Privileges assigned to roles are derived from responsibilities belonging to the attached job. Users are not permitted to perform any operation that is not part of their responsibilities. Roles and privileges are assigned and managed by administrators, not by users.

Resultant Context
This pattern provides an access control mechanism based on user responsibilities and organizational characteristics. *Access-Controlled-by-Roles*, and *Administrator* patterns help to complete this pattern.

Related Patterns
- *Dealing with Roles* [16] provides a whole pattern language discussing roles with more specific implementation details.
- *Roles* [17], Roles create one or more role objects that define the permission and access right for different user groups.
- *Administrator* pattern solves the user-role assignment problem.

Known Uses
- Some web servers use **.htaccess** and **.htgroups** files, which define groups of users (**Roles**) that can access certain areas of a web site.

- Oracle has a **Roles** feature for defining security privileges. User-role and role-privilege relationships are stored in tables.
- The PLoP 1998-registration program [18] has two **Roles**; attendee and administrator.

Administrator

Context
The *Role-Based-Access-Control* pattern introduces general roles, privileges, administrative roles and administrative privileges to design and manage a system.

Problem
In a RBAC environment the administrator is responsible for managing roles and privileges. In order to design the system with administrative capabilities, the following forces need to be balanced.

- When a new user enters the organization, an existing role or roles must be assigned based on his responsibilities.
- When a user leaves the organization or his responsibilities change, the user's current role must be revoked and, if needed, new role(s) must be assigned.
- Each role must be granted privileges based on the responsibilities and organizational security policies.
- In complex systems, there can be several administrators with different responsibilities. Some administrators may have privileges to assign, revoke and update roles and privileges, while others may assign roles but may not revoke or modify roles.

Solution

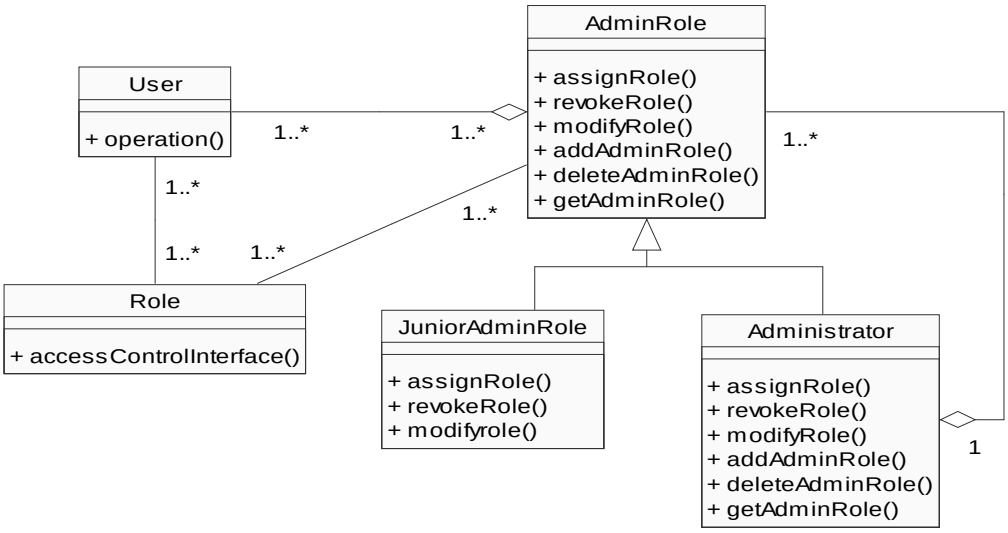

Figure 5. Solution structure

Participants

AdminRole defines the interface for administrator roles. It also defines the interface for handling junior administrator roles. AdminRole implements the default behaviour for all administrator roles, as appropriate. It needs a reference to a user object.

Administrator defines the behaviour of senior administrator roles, which controls junior administrator roles and user roles.

JuniorAdminRole defines the behaviour for junior administrator roles.

User represents the users who access the system.

Role defines the interface to restrict the method execution.

Collaborations
Administrator creates and deletes AdminRoles, and invokes JuniorAdminRole to assign, revoke or modify general roles to user objects.

Consequences
- Since JuniorAdminRole is under the control of Administrator, it can be restricted to handle only a particular set of users, roles or privileges. For example, a junior administrator may not be permitted to revoke roles assigned by another junior administrator or may only be permitted to assign a specific set of roles.
- JuniorAdminRoles can also be used to administer users and roles in local domains, in which case Administrator can centrally administer local administrative roles.
- Administrator can be implemented to create JuniorAdminRoles with different privileges.
- Since the *ProxyRole* pattern addresses the role-privileges assignment problem, this pattern addresses only the user-role assignment problem.

Resultant Context
Application of this pattern solves the user-role assignment problem. The *Access-Controlled-by-Roles* pattern helps to complete this pattern.

Related Patterns
- Administrator Manager components of **OASIS** [20] is a refinement of this pattern.
- This pattern uses the *Hook Method* pattern to implement administrative actions.
- *Proxy-Role* and *Role-Selection-Strategy* pattern solve the role-privileges assignment problem.
- This pattern uses the *Composite* pattern to design the administrative roles.

Known Uses
In **OASIS,** the administrators are responsible for certain security domains, they are allowed to access and manage the security information of these particular domains. The meta administrators are responsible for specifying domains, defining security concepts. The meta administrators and administrators are variants of the Administrators and JuniorAdminRoles of this pattern.

Implementation
- Since a user may have multiple roles, the User class may be implemented to hold a list of roles and their status. Administrator can be implemented to add, revoke and modify these roles.
- Administrator can be implemented to administrate both JuniorAdminRoles and other, common roles. JuniorAdminRoles are implemented to administrate common roles based on responsibilities. The methods assignRole(), revokeRole() and modifyRole() can be implemented to access User objects, and then assign, revoke or modify roles by manipulating the list of roles embedded in User objects. However, they should use different algorithms to administrate users and roles. *Template Method* or *Hook Method* patterns can be used to implement this functionality.
- One might use the *Command* pattern to implement assignRole(), revokeRole() and modifyRole() actions as objects.

Access Controlled by Roles

Context
The Role-Based-Access pattern introduces a set of common roles and privileges, and a distinct set of administrator roles and privileges. While the Administrator pattern addresses the user-role assignment problem, this pattern solves the role-privileges assignment problem.

Problem
After the administrator has assigned roles to users, the system must assess user requests on the basis of responsibilities (privileges) belonging to the users' roles. In most organizations, roles have overlapping responsibilities. Such common responsibilities can form role hierarchies. In order to support role hierarchies and role-privilege assignment, the following forces should be resolved:

- Users can execute operations attached to the responsibilities (privileges) of their roles; requests for unauthorised operations are denied.

Solution

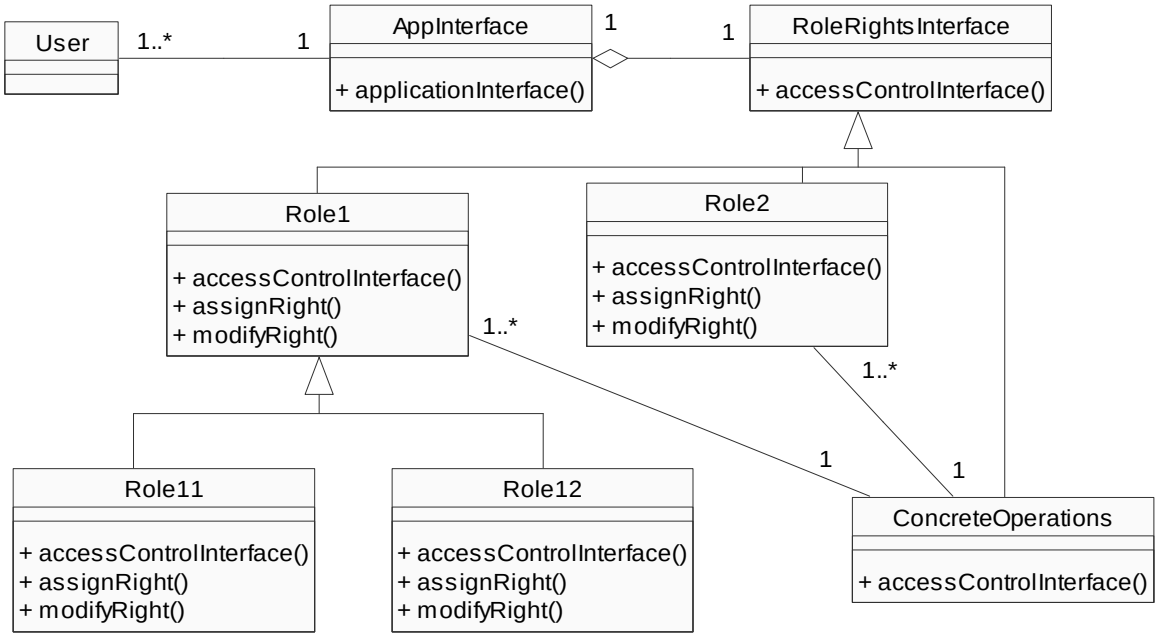

Figure 6. Solution structure

Participants

AppInterface defines the interface for user requests and creation of appropriate role objects for the users..

RoleRightsInterface defines the interface for common Roles and ConcreteOperations.

Role1, Role2, Role11, Role12 (Account holder, Employee, Teller, Account representative) define common roles; maintain reference to the ConcreteOperations and implement operations.

ConcreteOperations implement actual operations to access information in the account database.

Collaborations

- AppInterface forwards requests from users to an appropriate role objects. Usually, AppInterface creates an appropriate role object and passes requests to it.
- Role object interacts with ConcreteOperations to control method execution. If the requested method is authorized for that role, it is forwarded to the ConcreteOperation object. Unauthorized requests are turned down with an error message.

Consequences

- All operations are defined and implemented within the ConcreteOperations class. Since the implementation of the Role class is based on privileges, users cannot execute operations not part of their responsibilities.
- The actual functionality of each operation is implemented within the ConcreteOperations class, whereas role class implementation reflects the assignment of privileges (responsibilities and constraints).
- Hierarchies of RoleRightsInterface classes (role hierarchies) define a family of access control algorithms. Inheritance is used to prevent multiple implementations of common functionality; a senior role automatically has all roles junior to it in the hierarchy.

Resultant Context

Application of this pattern solves the role-privileges assignment. The *Role-Selection-Strategy, Proxy-Role***,** and Object-*Collection* patterns help to complete this pattern.

Related Patterns

- *Role-Selection-Strategy* selects the user's role and forwards request to that object.
- *Proxy-Ro*le controls the execution of methods.

Known Uses

- The **OASIS** [20] uses a variant of this pattern to model the security information base which supports users, roles, role-hierarchies, user-role assignment and authorization etc.
- This pattern controls the access to the information objects by restricting the execution of methods. *Method-based authorization* model [19] introduces a similar approach for object-oriented database systems.
- Implementation of RBAC [22] uses this pattern to control access to methods in protected information objects.

Object Collection

Context

The Role-Based-Access and Access-Controlled-by-Roles patterns control direct access to the information/data. Actual operations are designed as methods of a single class (ConcreteOperations). The methods are devolved into smaller classes.

Problem

In most cases the number of operations is fairly large. The ConcreteOperations object can be implemented as an aggregate of separate related objects. The following forces need to be resolved to design such a complex structure.

- Complex objects should either be decomposed into smaller objects or be composed of existing objects in order to have a flexible and extensible system.
- Decomposition of operations should be transparent to the user, i.e. users see every information object as a single unit.

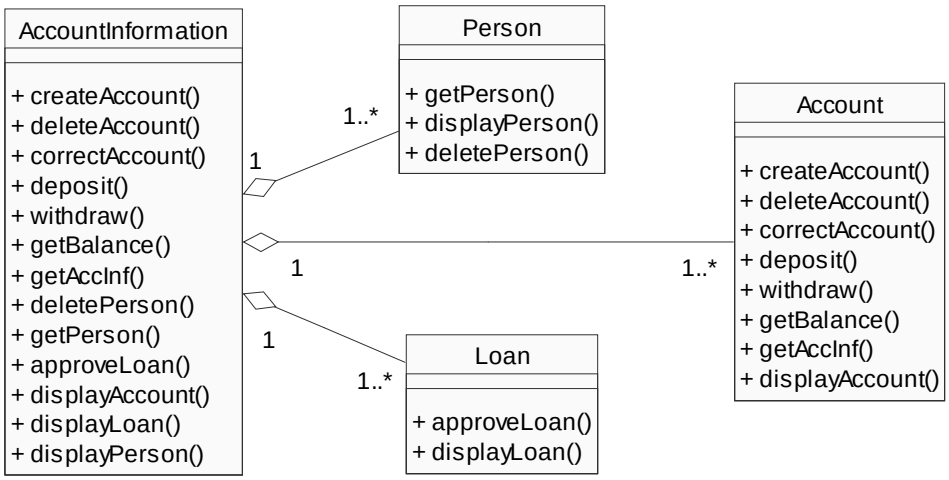

Figure 7. Structure of the access methods

Solution

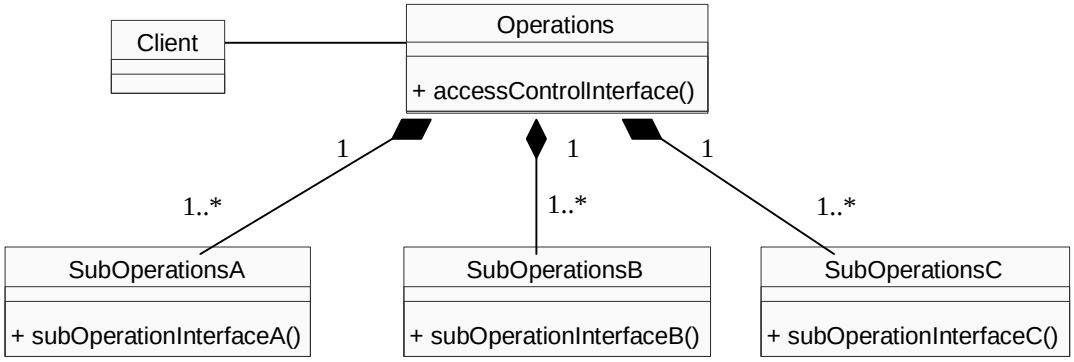

Figure 8. Solution structure

Participants

Operations is an aggregate of smaller objects each defining an interface that becomes part of the aggregate interface.

SuboperationA, SubOperationsB have only a particular set of operations declared in ConcreteOperations class.

Client requests a service from CobcreteOperations object.

Collaborations

ConcreteOperations provide services made of smaller objects. It translates a request to execute a particular operation into calls of the relevant smaller objects.

Consequences

Only operations of ConcreteOperations are visible to the users, the smaller objects remain hidden.

Resultant Context

Application of this pattern transforms complex structures into a set of simpler structures.

Related Patterns

- *Whole-Parts* pattern also structures complex structures into a collection of small structures.
- *Composite* pattern is applicable for representing whole-part hierarchies when clients should ignore differences between compositions and individuals.
- *Façade* provides a simple interface to a complex subsystem. However, *Façade* does not force encapsulation of the components.

Known Uses

- Most object-oriented class libraries provide collection classes such as lists, vectors, sets and maps. Implementation of these classes uses the *Whole-Part* variant of the *Object-Collection* pattern.
- Most of the graphical user interface toolkits use the *Composite* variant of the *Object-Collection* pattern.

Implementation

- Operations can be implemented to invoke appropriate smaller objects.
- The SubOperations class should be implemented to perform the exact operations. The Operation class can also be implemented by inheriting all the SubOperation classes.

Proxy Role

Context
The Role-Based-Access and Access-Controlled-by-Roles pattern control direct access to the information/data. In these patterns, method execution is controlled by roles. The Proxy-Role pattern complements the Access-Controlled-by-Roles pattern.

Problem
According to the Access-Controlled-by-Role pattern, a representative, a role, will access the information on behalf of the user. The question is: *How can you design these roles with restrictions to method execution?*

Solution

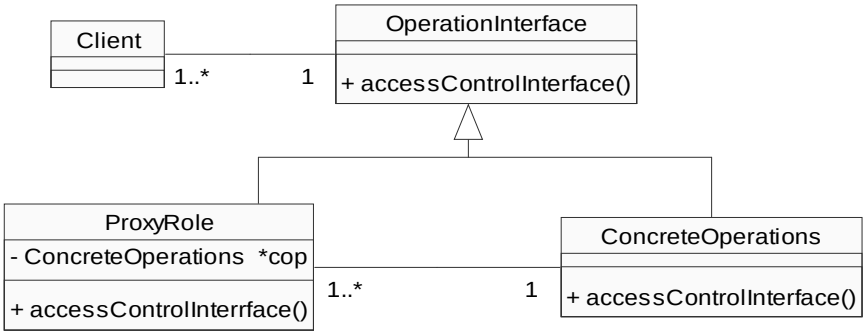

Figure 9. Solution structure

Participants

Client interacts with the OperationInterface.

OperationInterface defines a common interface to the ProxyRole and ConcreateOperations objects.

ProxyRole implements the Interface based on authorized privileges. ProxyRole maintains a reference to the ConcreteOperations object.

ConcreteOperations implements the actual functionality of the Interface.

Collaborations
Client requests the service from OperationInterface. Then OperationInterface forwards the request to a ProxyRole object. ProxyRole evaluates the received request, and if the role's privileges allow the operation, the request is forwarded to ConcreteOperations. Otherwise the request is rejected.

Consequences
- Users can request the execution of any operation regardless of their privileges, but the role object prevents the execution of unauthorized operations. In case of unrelated roles each role can be designed as a ProxyRole object with different functionality. Inheritance hierarchies can be used to design role hierarchies.
- Since the implementation of role objects are based strictly on privileges, this pattern simplifies the administrative task by solving the role-privilege assignment problem.

Resultant Context
Role-Selection-Strategy pattern helps to design role hierarchies.

Related Patterns

- *Full-View-With Errors* [17] designs application so user can see that they might have access to. It validates the user request to perform operations and users are notified with an error message when they request to perform illegal operations.
- *Proxy* pattern provides an indirection to access information. In particular, *Protection Proxy* pattern protects information from unauthorized access
- *Role-Selection-Strategy* uses this pattern to design roles and role hierarchies.

Known Uses

- SQLPlus, used to access Oracle database, allows users to execute any operation and displays an appropriate error message if illegal access is attempted.
- Most of the word processors and text editors let the user try to save over a read-only file but displays an error message after the save has been attempted.
- OMG-CORBA uses the Proxy variant of the ProxyRole patterns for two purposes. The client-stub guard clients against the concrete implementation of their servers and the ORB. ORB uses the IDL-skeleton to forward requests to concrete remote server components.
- The Java programming model of throwing exceptions follows this pattern. Instead of designing methods to only be used in the proper context, exceptions are thrown forcing the rest of the application to deal with the error.

Implementation

- The OperationsInterface class defines the interface for roles and operations defined in the system.
- The ConcreteOperations class should be implemented to access information objects and to represent the exact functionality of the operations declared in OperationsInterface.
- Implementation of the role class is strictly based on privileges. Authorized methods may be implemented to call the corresponding methods in the ConcreteOperation class, and unauthorized operations should be implemented as request rejections.

Role Selection Strategy

Context

This pattern supports role hierarchy design, finding out the user's role and forwarding requests to an appropriate role object.

Problem

The question is: *How can you design the system to balance the following forces?*

- Users are allowed to make any request but requests can be accepted or rejected based on the user's role.
- When a user requests the execution of an operation, the request should be forwarded to the correct role object.
- User access to information is based on responsibilities and organizational security policies.

Solution

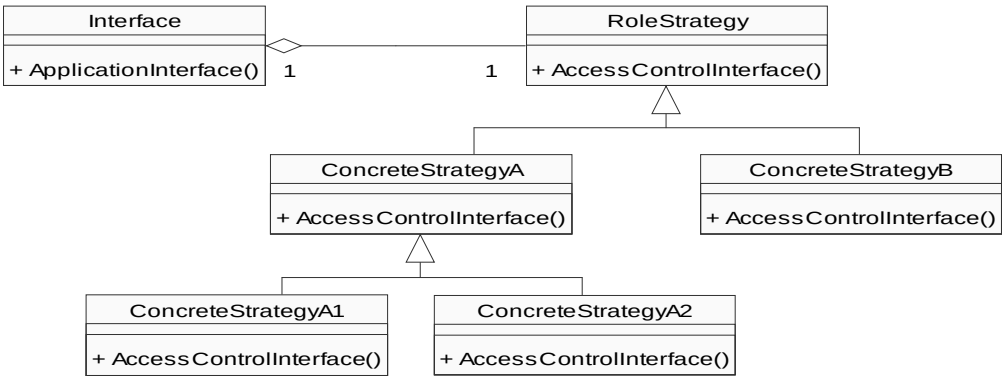

Figure 10. Solution structure

Participants

Interface defines role selection strategy and is configured with a ConcreteRole object. It maintains a reference to a RoleStrategy object.

RoleStrategy declares an interface common to all roles. Interface uses Role to call an access control algorithm defined by a ConcreteRoleStrategy.

ConcreteRoleStrategy (Account-Holder, Employee, Teller etc.) implement accesses control algorithms and use the RoleStrategy interfaces.

Collaborations

- User enters his identification and a role name and requests operations to be executed. Interface object checks if the user has been assigned that role. If the answer is yes, it creates an object of the given role-type and forwards the request. Otherwise it turns down the request.
- When a role object has received a request, it executes the authorized methods and returns the result. Unauthorized methods are rejected and a warning message is returned.

Consequences

RoleStrategy classes define a family of role hierarchies. Inheritance can be used to rationalise common functionality in access control algorithms.

Related Patterns

- *Sponsor-Selector* pattern [14] provides a mechanism for selecting the best resource for a task from a set of resources that change dynamically.
- *Role-Selection-Strategy* pattern uses the *Strategy* pattern to forward user requests to an appropriate role object.
- *Template Method* pattern defines the skeleton of an algorithm, deferring some steps to subclasses.
- *Role-Selection-Strategy* pattern uses the *Hook Method* pattern to implement strategy classes.

Known Uses

- SQL code generator [21] uses the Strategy variant of this pattern.
- ET++: Object- Oriented application framework in C++ uses Strategy variant to encapsulate different line breaking algorithms.
- Implementation of RBAC [22] uses this pattern to forward user requests to an appropriate role object.

Implementation

Interface class might be implemented as:

```
class Interface {
public:
    enum RoleType { AccountHolder, Employee, Teller, AssistantManager, Manager,  InternalAuditor,
    AccountRepresentative, FinancialAdvisor };
      Interface()        { role = NULL; }
     ~Interface() {;}
     void SetRoleObject( string rolename );
     void ExecuteOperation();
private:
      Role*  role;
};
```

- The setRoleObject() method can be implemented to read userID and rolename. If the user has the specified role, the new object can be created.
- ExecuteOperation() method might be implemented to read the account number, to provide an interface for method selection, to create an appropriate role object and to invoke the selected method on the role object.

- The RoleStrategy class can be defined as an abstract class to provide a common interface to all access control algorithms or it may implement the default behaviour of the algorithm.
- ConcreateRoleStrategy class can be implemented by overriding methods of the RoleStrategy class. For example, AccountHolder and Teller have different privileges to execute the getAccout() method, so, both of them can override the getAccount() method with its own functionality.

When there are role hierarchies, senior roles can be implemented by overriding the necessary methods. For example, Employee implements the createAccount, deleteAccount and correctAccount methods to produce warning messages. AccountRepresentative can be implemented by overriding these methods to perform those operations.

4. Discussion

This work combines and composes design patterns into a pattern language for developing Role Based Access Control systems. The six design patterns address the different phases of role and access privileges assignment and maintenance in RBAC. The connections between the patterns are shown in figure 11. Each oval represents a pattern, and arrows represent the relationships between patterns.

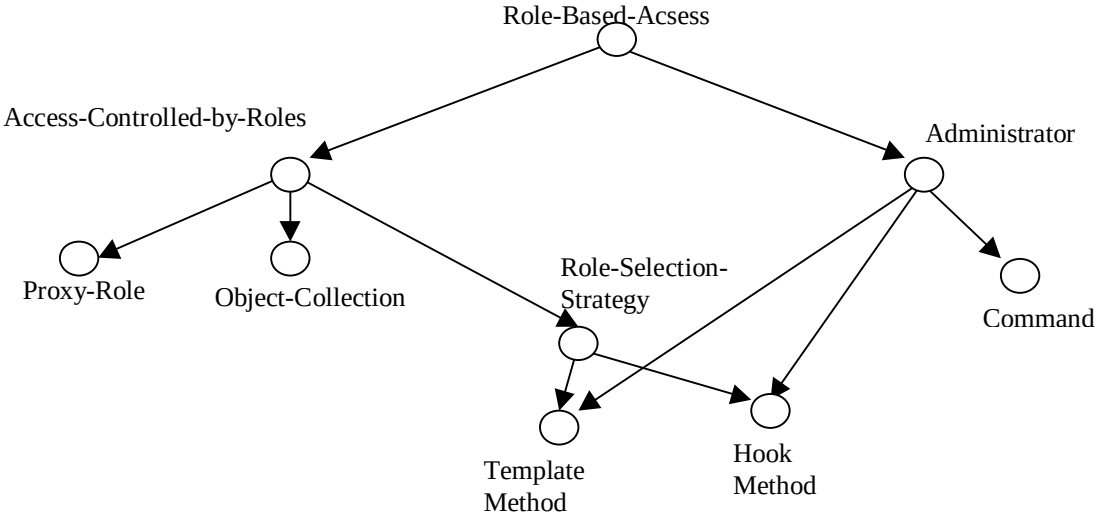

Figure 11. Structure of the language

The *Role-Based-Access* pattern introduces a set of roles and a set of privileges. In the next step, the *Acces-Controlled-by-Roles* pattern is used to design those roles that will eventually control access to the protected RBAC objects. The *Proxy-Role, Role-Selection-Strategy* and *Object-Collection* patterns form the *Access-Controlled–by-Roles* pattern by designing roles, role hierarchies and complex operation structures respectively. The *Administrator* pattern completes the *Role-Based-Access* pattern by designing administrative roles to handle user-role assignments and to delegate administrative responsibilities.

The *Proxy-Role, Object-Collection* and *Role-Selection-Strategy* patterns together implement the access control mechanism. The ProxyRole class controls the execution of methods: requests consistent with roles are forwarded to the corresponding methods in ConcreteOperations class and other requests are rejected. The ConcreteOperations class implements the operations on protected RBAC objects. The Interface class creates appropriate role objects for users and forwards user requests to those objects. Each ConcreteRoleStrategy class overrides the RoleStrategy interface according to its privileges, i.e. the implementations of role classes perform the role-priviles assignment.

The Administrator class implements user-role assignment. It also includes methods that handle local administrators in complex systems. Some administrative responsibilities are delegated to the JuniorAdminRole class that implements user-role assignment subject to local restrictions.

Applications using access control are usually based on a fixed set of routinely performed operations. The pattern language presented here defines a complete set of such operations and controls access to them by introducing roles. Applications of this language have the following benefits and liabilities.

- For each role of a user the AppInterface object creates a separate role object. When access rights associated with a role are changed, the changes can easily be propagated to these role objects.
- Access rights are located exclusively within the role classes. Hence any changes to the policy describing access rights will affect the role classes only.
- The pattern language introduces administrator roles for user-role assignments. Since the privileges are implicitly assigned to roles through the implementation of role classes, administrator can assigned appropriate roles to users based on user responsibilities.

5. Conclusions

- Object oriented design patterns capture many of the features in a RBAC environment. In particular, objects and associated methods can be used to define protected RBAC objects and operations, and method overriding can be used to implement roles and role hierarchies.
- The pattern language presented can be applied to design and implement RBAC systems, and a change in role policy has a very limited effect on the whole system.
- The administration of roles and privileges implemented by OO design patterns is easy and flexible.

Acknowledgments

We would like to thank our KoalaPLoP shepherd Eduardo B. Fernandez who guided us through several iterations of the article and provided many constructive comments.

References

[1] C. Alexander, The *Timeless Way of Building*, Oxford University Press, first edition, 1979.
[2] C. Alexander, M. Silverstein, and S. Ishikawa, *A Pattern Language*. Oxford University Press, first edition, 1977.
[3] F. Buschmann, R. Meunier, H. Rohnert, M. Stal and P. Sommerland, *A System of Patterns*. Addition-Wesley, first edition. 1996.
[4] J. Coplien, *Software patterns* a white paper. SIGS publications, 1996.
[5] J. Coplien, and D. Schmidt, *Pattern Languages of Program Design*, Addition-Wesley, first edition, 1995.
[6] D. F. Ferraiolo and D. R. Kuhn, *Role-Based Access Controls*, Proceedings of the 15th NIST-NSA National Computer Security Conference, Baltimore, Maryland, October 13-16, 1992.
[7] D. F. Ferraiolo, J. A. Cugini, and D. Richard Kuhn, *Role-Based Access Control (RBAC): Features and Motivations*, 11th Annual Computer Security Applications Proceedings, 1995.
[8] Department of Defense, *Trusted Computer Security Evaluation Criteria*, DoD 5200.28-STD, 1985.
[9] M. Fowler, *Analysis Patterns reusable object modules*. Addition-Wesley, first edition, 1997
[10] E. Gamma, R. Helm. R. Johnson, and J. M. Vlissides,, *Design patterns elements of reusable object oriented software*. Addition-Wesley, second edition, 1994.
[11] J. M. Vlissides , J. Coplien, and N. L. Kerth, *Pattern Languages of Program Design 2*. Addition-Wesley, first edition, 1996.
[12] W. Pree,, *Design Patterns for Object Oriented Software development*. Addition-Wesley, first edition, 1994.
[13] R. S.Sandhu., J. C. Edward, L. F. Hal, and E. Y. Charles, "Role-Based Access Control Models", IEEE Computer, Vol. 29, February 96, pp 38-47.
[14] D. Riehle, R. C. Martin, F. Buschmann, *Pattern languages of program design 3*. Addition-Wesley, first edition, 1998.
[15] L. Zhao, and T. Foster, *Plots: A Pattern Language of Transport Systems- Point and Route*. Pattern Language of Program design 3, 1998.
[16] M. Flower, *Dealing with roles*: Collected papers from the PLoP97 and EuroPLoP 97 Conference, Technical Report #wucs-97-34, Dept. of Computer Science, Washington University Department of Computer Science, September 1997.
[17] N. Harrison, B. Foote, and H. Rohnert, *Pattern Languages of Programs Design 4*. Addison Wesley, 2000.
[18] J. Yoder and D. Manolescu, *The PLoP Registration Framework*, 1998. URL: `http://www.uiuc.edu/ph/www/j-yoder/Research/PLoP`.
[19] E. B. Fernandez, M. M. Larrondo-Petri and E. Gudes, *A method-based authorization model for object-oriented databases*, Proc. Of the OOPSLA 1993 Workshop on Security in Object-Oriented Systems, 70-79.
[20] W. Essmayr, E. Kapsammer, R. R. Wangner, G. Pernul and A. M. Tjoa, *Enterprise-wide security administration*, Procs. 9th Intl. DEXA Workshop, 1998, 267-272.

[21] K. Brown, *Using the Strategy and Command patterns for SQL code generation*. URL: http://hometown.aol.com/kgb1001001/Articles/Command/CommandJava.htm
[22] J. Barkley, *Implementing Role Based Access Control Using Object Technology*, First ACM Workshop on Role-Based Access Control
[23] E. G. Amoroso, Fundamentals of Computer Security Technology. Prentice hall, 1994