CSE 367 CS Research Lab

Prof. Steve Demurjian

Final Project Report

14 May 2001

Implementation of Constraint-based Security Model in JINI Environment

Group Members:

Qi Jin (Qi): qjin@engr.uconn.edu

Mumtaz Lohawala (Mumtaz): mumtaz@engr.uconn.edu

Nayana A Limaye (Nayana): nsp@engr.uconn.edu

Table of Contents

1. Introduction and Motivation

Security for a distributed environment is an ever-increasing problem, and seeking solutions for web-based, distributed, and agent-based applications is critically important, and has been investigated from many different perspectives. In the past 5 years, role-based approach to security for object-oriented models has customized the public interface, allowing different subsets of the public methods to be visible to different users based on role. By cataloging these different subsets of methods across an entire class library, it is possible to enforce, at runtime, this custom behavior, in object-based, agent, and distributed settings. Thus, an end user playing a role via a client application can be restricted as to which methods can be invoked, and the same client application will consequently work different at different times and different locations based on role.

In the previous semester, we incorporated the use of a security token to build a JINI-based security system. This system could restrict the end-users' access to the system according to their user name/password, their roles and their locations (IP address). However, our system had the following major limitations:

1. From the user-role perspective, the user was either simply granted or denied the ability to play certain roles;
2. From the role-method call perspective, the role was either simply granted or denied the method call;
3. There were no tracking and analysis tools to keep track of the activities going on in the system;
4. The interface of the security system was not user-friendly.

This semester we, therefore, strove for a more versatile and robust system by extending our existing security model to incorporate a solution to the above limitations. The major extensions made this semester to increase the scope of the existing security model are the addition of "time and signature constraints" and the inclusion of a "tracking and analysis tool".

The two types of constraints, viz., *Time constraints (TC)* and *Signature constraints (SC)* limit the accessibility of system resources by users depending on the time at which the user plays a particular role and the actual role that he plays. *Time*

Constraints means that the action has to be performed within valid time intervals. *Signature Constraints* means that the roles' invocation of a method is based on allowable values. These constraints control method invocations by limiting the time period of the invocation (time constraints) and are based on the return and/or parameter values (signature constraints). Signature constraints are independent of database integrity constraints (which are the responsibility of the resource) and programming types (which are the responsibility of the compiler/runtime environment). In our current version of the security system, users can play roles based on *TC*, and roles can access methods based on *TC* as well as *SC*.

As an enhancement, the security system will keep a dynamic tracking log to let security officers trace all the activities occurring in the whole environment. Likewise, a static analysis tool is employed to guarantee that the inner method calls are consistent with the security policies.

To make the system more intuitive for the security officers, GUI improvements were done to the existing security policy and authorization lists. GUI modifications were also done to the resource system to prevent illegal accesses by users to the system hence making the system secure right from the log-in time.

In the remainder of this report, we will first introduce our new system architecture, especially our improvements to the security system in section 2. Then we will illustrate the logic and implementation of the two core parts of the security system viz., the client (user) registration and the constraint-based authorization in section 3. In section 4, we will discuss the implementation of signature constraints and time constraints. We will concentrate on the implementation of dynamic tracking and static analysis tools in section 5. Finally, in section 6, we will conclude our work done in this semester and propose some issues and enhancements for the future system. In the appendices, we will include a user's manual and UML diagrams of the current system.

2. Software Architecture and Improvement

In Figure 1, we represent the security-related clients and resources that comprise our new system architecture. We have a Unified Security Resource (USR), which provides role-based security support in Distributed Resource Environment (DRE). USR provides four categories of services:

1. Security Policy Services define user roles (UR's), register the resources, services and methods, and grant access to roles for resources, services and/or methods. Security Policy Services also include several methods that are provided for resource servers to publish themselves, their services and methods.
2. Security Authorization Services are utilized to maintain profiles on the clients (e.g. users, tools, software agents, etc.) that are authorized and actively utilizing non-security services. These services allow a security officer to grant users to roles.
3. Security Registration Services are utilized by clients at start-up time for identity registration (User id, IP address and user role).
4. Security Tracking and Analysis Services are utilized by the security officers to dynamically trace the activities in the security environment and statically analyse the inner method calls of the resource server.

In addition to USR, there are two security clients bundled with the security resource: a *Security Policy Client (SPC)* and a *Security Authorization Client (SAC). SPC* enables the security officer to manage user roles by granting/revoking privileges (resource, service, and/or methods) to/from user roles and to support tracking and analysis of defined security privileges. *SAC*, on the other hand, enables the security officer to authorize roles to end users, and to assign them negative and additional privileges. A *Global Clock Resource*[1] *(GCR)* is introduced to support time-constrained access of resources (and their services and methods) by the user role. An *Analysis Tool Resource* is provided for security officers to statically analyze the inner method calls of the resource server and keep the consistency of the inner method calls with the security policies.

[1]We didn't implement GCR this semester. We only put it in the conceptual model of this report.

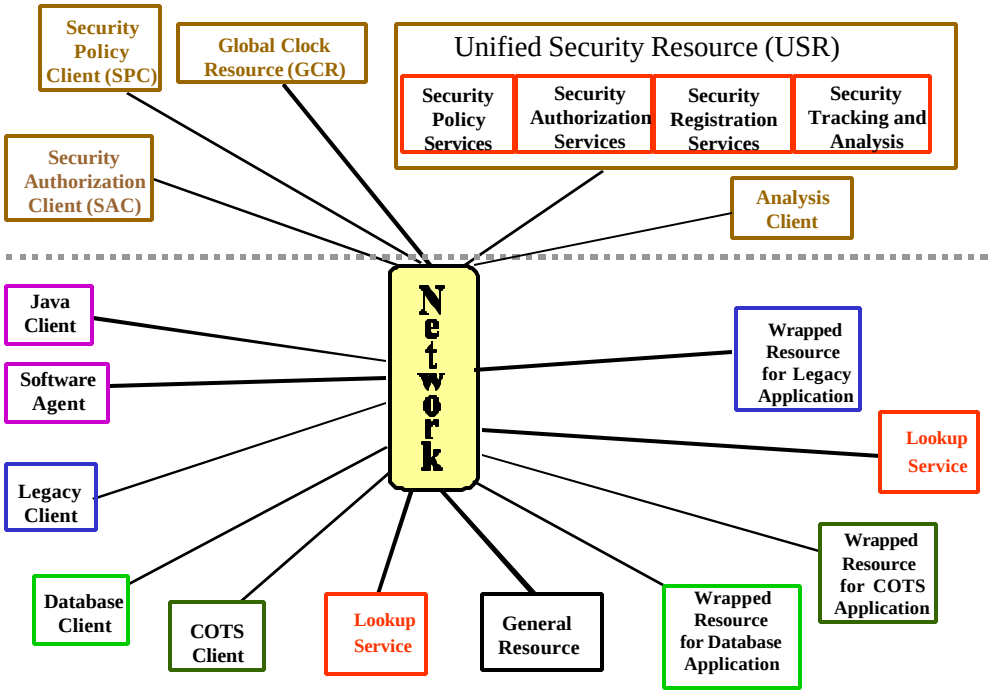


Figure 1: Security Clients and Resources in Distributed Environment.

The prototype of our system has a USR and a resource system, the Course Database Resource. The Course Database Resource comes with a server and a client. In the server, we have two services: Read Services and Modification Services. Each of these services has a couple of methods. Please refer figure 2 for the list of services provided by the resource server. The Course Database Resource Client is used to access USR for registering clients and accessing Course Database Server for actions. The focus of this report will be on USR, and all the different interactions that take place with resource and clients in dynamic environment. In Section 2.1, we will list the services and methods of the USR. In Section 2.2, we will present the token-based interaction within the DRE environment. Section 2.3 will consider the process from the perspective of the resource registering itself, its services and its methods for a secure access.

2.1 The Unified Security Resource (USR)

The Unified Security Resource (USR) is a repository for all static and dynamic security information on roles, clients, resources, authorizations, etc., and is organized into a set of services, as given in Figure 2.

SECURITY POLICY SERVICES

Register Service
```
Register_Resource(R_Id);
Register_Service(R_Id, S_Id);
Register_Method(R_Id, S_Id, M_Id);
Register_Signature(R_Id, S_Id, M_Id, Signat);
UnRegister_Resource(R_Id);
UnRegister_Service(R_Id, S_Id);
UnRegister_Method(R_Id, S_Id, M_Id);
Unregister_Token(Token)
```

Query Privileges Service
```
Query_Resource();
Query_Method(R_Id);
Query_MethodDesc(Token, R_Id, S_Id, M_Id);
Check_Privileges(Token, R_Id,
                 S_Id, M_Id, ParamValueList);
```

User Role Service
```
Create_New_Role(UR_Name, UR_Disc, UR_Id);
Delete_Role(UR_Id);
Query_Role(UR_Id)
```

SECURITY AUTHORIZATION SERVICES

Authorize Role Service
```
Grant_Role(UR_Id, User_Id);
Revoke_Role(UR_Id, User_Id);
```

Client Profile Service
```
Verify_UR(User_Id, UR_Id);
Query_Client(User_Id);
Erase_Client(User_Id);
Find_Client(User_Id);
Find_All_Clients();
```

Constraint Service
```
DefineTC(R_Id, S_Id, M_Id, SC);
DefineSC(R_Id, S_Id, M_Id, SC);
CheckTC(UR_Id, R_Id, S_Id, M_ID,
CheckSC(UR_Id, R_Id, S_Id, M_ID, ParamValueList);
```

Grant-Revoke Service
```
Grant_Resource(UR_Id, R_Id);
Grant_Service(UR_Id, R_Id, S_Id);
Grant_Method(UR_Id, R_Id, S_Id, M_Id);
Grant_SC(UR_Id, R_Id, S_Id, M_Id, SC);
Grant_TC(UR_Id, R_Id, S_Id, M_Id, TC);
Revoke_Resource(UR_Id, R_Id);
Revoke_Service(UR_Id, R_Id, S_Id);
Revoke_Method(UR_Id, R_Id, S_Id, M_Id);
Revoke_SC(UR_Id, R_Id, S_Id, M_Id, SC);
Revoke_TC(UR_Id, R_Id, S_Id, M_Id, TC);
```

SECURITY REGISTRATION SERVICES

Register Client Service
```
Create_Token(User_Id, UR_Id, Token);
Register_Client(User_Id, IP_Addr, UR_Id);
UnRegister_Client(User_Id, IP_Addr, UR_Id);
IsClient_Registered(Token);
Find_Client(User_Id, IP_Addr);
```

SECURITY TRACKING AND ANALYSIS SERVICES

```
Tracking Service
Logfile(Log String)

Analysis Service
Analyze(Java Class File)
```

Figure 2: The Services of USR.

Security Policy Services are utilized to define, track, and modify user roles, to allow resources to register their services and methods (and signatures), and to grant/revoke access by user roles to resources, services, and/or methods with optional time and signature constraints. These services are used by a security officer to define a policy and by the resources (e.g., database, Java server, etc.) to dynamically determine whether a client has the permission to execute a particular [resource, service, method] under a time and/or signature constraint. There are five different security policy services: **Register** for allowing a resource to (un)register itself, its services, and their methods (and signatures), which are used by a resource for secure access to its services; **Query Privilege** for basic introspection and verification of privileges, used by the security officer via the security policy client and at runtime by the resource to verify whether the client (via a token) has the permission to invoke a method under a set of constraints; **User Role** to allow the security officer to define and delete user roles; **Constraint** to allow time and signature constraints to be defined by the security officer, and for these constraints to be dynamically verified at runtime; and, **Grant-Revoke** for establishing privileges, where a UR can be granted/revoked a resource, its services, and their methods,

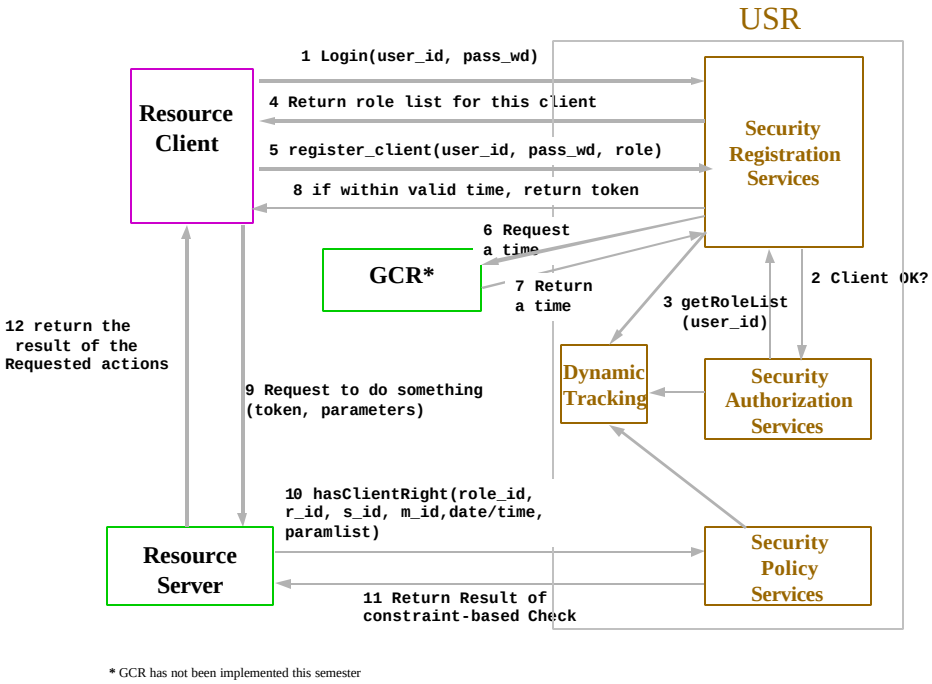
with or without time and signature constraints. The security officer can access these services from Security Policy Client.

Security Authorization Services are utilized to maintain profiles on the clients (e.g., users, tools, software agents, etc.) that are authorized and actively utilizing non-security services, allowing a security officer to authorize users to roles with or without time constraints. There are two services: **Authorization Role** for the security officer to grant and revoke a role to a user with the provision that a user may be granted multiple roles, but must play only a single role when utilizing a client application; and, **Client Profile** for the security officer to monitor and manage the clients that have active sessions. The security officer can access these services from Security Authorization Client.

Security Registration Services are utilized by clients at start-up time for identity registration (client id, IP address, user role and access time), which allows a unique token to be generated for each session of a client. The single service, **Register Client**, can also be utilized by the resource to decide whether the user can play the role, verify client identity and by the security officer to monitor client activity via the security authorization client.

Security Tracking and Analysis Services are utilized to dynamically track all the activities in the security environment and to statically analyze the inner method calls of the resource system. There are two services: **Tracking Service**, which maintains a real-time log file for the security officer to trace and monitor all the successful and unsuccessful activities (including user sign-in/sign-out, the status of user's access to resource/service/method, the reason for access failure or success, etc) in the security environment. **Analysis Service**, which will help the security officers to analyze the inner method calls of the joining resource server and keep the consistency of actual method calls and the security policy. The security officer will use Tracking Service from Security Policy Client and Analysis Service from Analysis Client.

2.2 Security Enforcement for Executing Client Application



* GCR has not been implemented this semester

Figure 3: Client Interactions and Service Invocations.

The processing required by a client application that is joining the distributed environment and subsequently attempting to access resources, is best illustrated with an example, which is given in Figure 3. In the first step in the process, the Course DB Client must be authenticated to verify that the client has access to the desired resource. Then he selects the specific user role that he wants to play in this session. After this, a token is generated and assigned to the user for the session. To do so, the Course DB Client first enters his user name and password. USR verifies his identity and returns him the roles he can play in the current resource system. The Course DB Client then picks a role and registers with USR via the Register_Client method, which must verify that the user is in the valid time interval for playing this role. USR then requests a global time from GCR[1] and then returns a token to the client via the Create_Token method. A new token is generated each time a valid user enters the system and begins a session. Since GCR processes all requests sequentially, a unique time is always returned, and hence the token that consists of User-ID, Role-ID, IP address, and creation time, is always unique. Even if the user has multiple sessions open at the same time from the same machine with the same role, the creation time of the token distinguishes the sessions. Assuming that the

registration and token generation were successful and Course DB Client finds the desired method from lookup service (can be JINI lookup service or CORBA lookup service), the user can then attempt to utilize resources from the distributed application, and in this example, from the Course DB Resource. In the example of Figure 3, a proxy to the method is returned, and the method is invoked with the parameters - Token and the parameters required by the specific method. The Course Database Resource has two critical steps to perform before executing RegisterCourse. First, CourseDB Resource verifies that the Client has registered with the security services. If this fails, a negative result is sent back via the method invocation result. If this is successful, then the Course Database Resource must perform a constraint-based check to verify if the user role can access the method limited by signature constraints and/or time constraints (both may be null), the Course Database Resource will return the result of the requested invocation to the client. The logic of Client (User) Authentication and Constraint-based Authorization will be explained in Section 3. All the activities in USR will be tracked dynamically by the dynamic tracking tool in USR.

2.3 Participation of Resources in Security Paradigm

In our enforcement framework as presented in Figure 1, there is an implied requirement that those resources that want to utilize the security capabilities, must publish themselves to USR. Specifically, each resource must register with the Security Policy Services, so that the master resource list, service list, and method list (and their signatures), can be modified to include resources as they dynamically enter the environment. Resources must be allowed to both register and un-register themselves, their services, and methods (with signatures) via the Register service. For our example, the Course Database Resource that we have utilized would need to register its Read and Modification services and all of their methods and signatures. This registration is critical, since it stores the relevant information via USR into a relational database system, which is then accessible to a security officer using either the Security Policy Client or the Security Authorization Client to establish and review security requirements. This process is illustrated in Figure 4, with the enumeration of all of the register and unregister methods. USR provides a set of methods to help the resource register or update its own profile. Whenever the resource server is online, it will use the methods provided by USR

to register (if this is the first time it is online) or update (if it is registered before) its own profile in the database of security system. The security officer can always unregister any resource/service/method from Security Policy Client at any time.

In addition to this process, as we discussed in Section 2.2, the Course DB Resource uses the USR at runtime to dynamically verify if the client is authorized to a [resource, service, method] under optional time and security constraints.

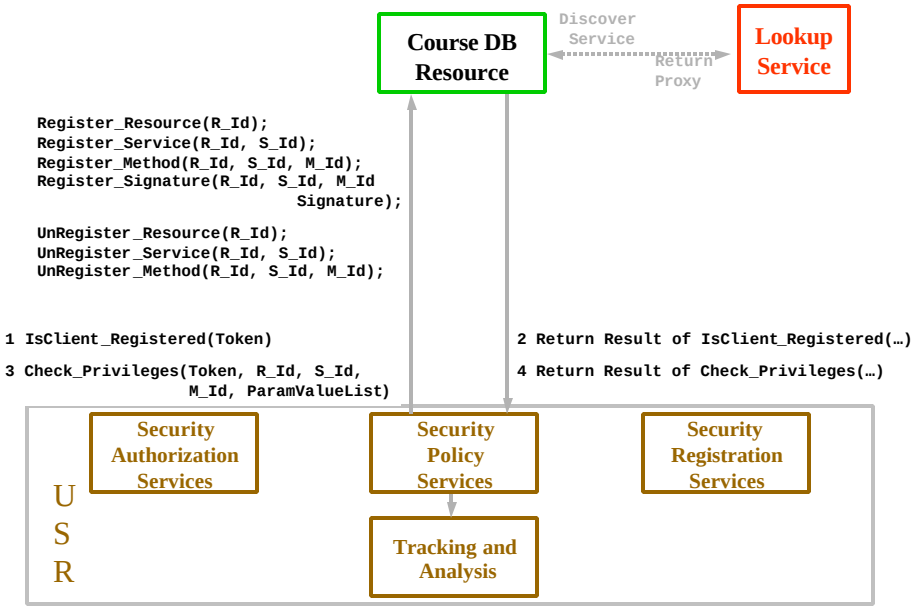


Figure 4: Resource Interactions and Service Invocations.

3. Implementation of Client (User) Registration and Constraint-based Authorization

As we mentioned in Section 2, the core parts of USR are Client (User) Registration and Constraint-based Authorization. In Section 3.1, we will discuss the logic of Client (User) Registration and in Section 3.2, we will discuss the logic of Constraint-based Authorization.

3.1 Client (User) Registration

This part of the security system verifies the identity of the user at login time before the user can be allowed to access the resource that he is trying to access. Following are the steps involved in client registration/authentication (and figure 5 depicts

this process in the form of a flowchart):

1. User must input the user id and password first.

   Error: Incorrect user id or password or both.

2. If the user id and password are valid then a role list is brought to the user from which he can select the role that he wants to assume. This role list is the list of roles that this user can possibly have for the given resource.

   Error: The user cannot assume any role for the given resource.

3. After he selects a role, the client will register with USR via the Register_Client() method which will verify whether the user is in the valid time interval to play this role and then return a token via the Create_Token() method.

   Error: The user cannot access the resource by this role in invalid time interval.

Here we give an example to illustrate the Client (User) Authentication process. Qi is a user of Course Database System. When she starts a session, the system will first prompt her to input her user name and password. If both the user name and password are correct, she will be asked to select a role from the roles she can play in Course Database System. Qi then selects CSEUndergrad role. Once she submits, USR then checks if she is allowed to play this role within the valid time interval. If so, USR finishes authentication for her in this session and returns her a unique token to access Course Database System.

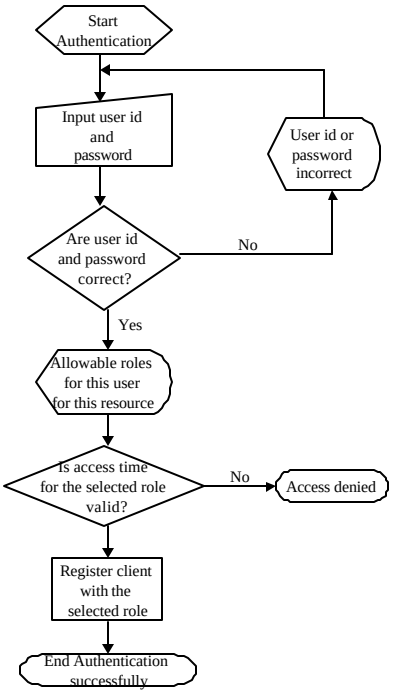


Figure 5: Flowchart for Client (User) Authentication

3.2 Constraint-Based Authorization

If the user passes the authentication check above, he can use the services offered by the resource. However, in order to be able to use any service, the user must have the right to use it. The privileges to access resources and their services (methods) are determined by a policy, which is dictated by a Security Officer. A user can access a service (or method) if all of the following are satisfied:

1. Token is valid
2. IP is valid
3. IP constraints are met with
4. No negative privilege is granted (to the user) to access that method[2]
5. Positive privilege is granted (to the role) to access the method
6. Time constraints are met with, i.e. the access is within the valid time interval
7. Signature constraints are met with (given to the role) if there are any signature constraints defined for the method

A flowchart for this kind of constraint-based authorization is given below:

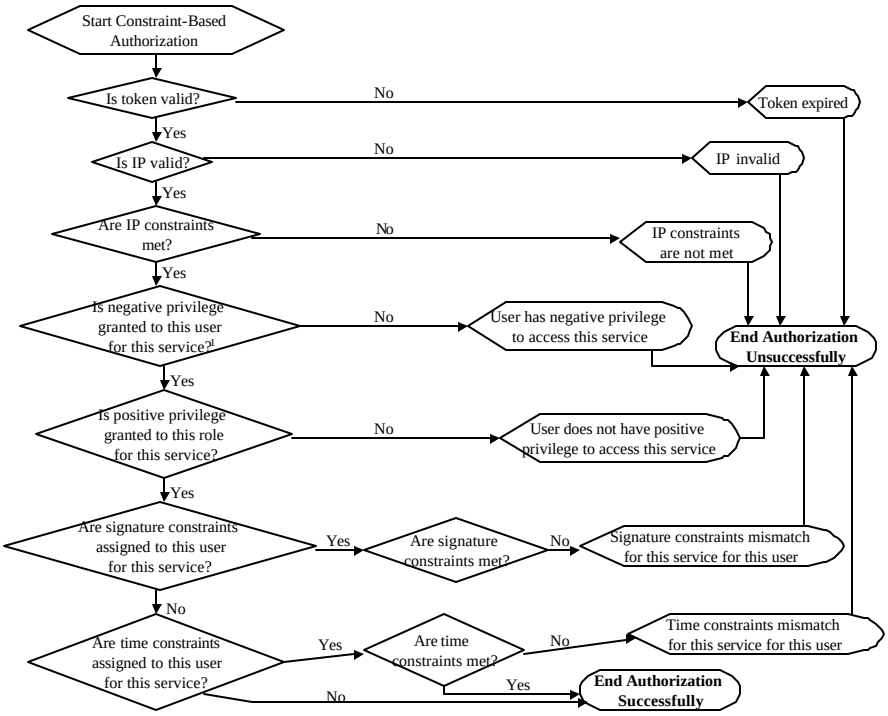


Figure 6: Flowchart for Constraint-Based Authorization

The order followed while checking the above privileges and constraints is 1 to 7 (above). If an error is encountered at any of these checks then the user cannot access that service (or method). These checks are described below in detail:

Token Check

This check is performed to verify the validity of the token to ensure that the token has not expired. However, according to Prof. Demurjian's feedback, we now set the token expiration time to be infinite.

Error: The token has expired.

[2] In the new security model, we will not have a negative privilege check. We have, however, not removed it from the code in case we might need it later. But since there are actually no negative privileges defined in the database, the constraint-based authorization just skips it.

IP Constraint Check

This check is done in order to ensure that the user logs on to the system from the permitted locations.

Error: The user cannot log on to the system from this location.

Negative Privileges Check

The negative privileges, given to a user, on either a whole resource or a service (method), disable that particular user from accessing that resource or service (method) respectively.

Error: The user has a negative privilege for either the whole resource or the service (method) that he is trying to use in the given resource.

Positive Privileges Check

The positive privileges are assigned to a client (role) in order to enable him to use a resource or its methods.

Error: The client (role) does not have a positive privilege for the service (method) that he is trying to use in the given resource.

Signature Constraints Check

Signature constraints allows to restrict the use of methods by clients (roles) in the system depending upon the value of the signature passed as arguments and returned as return value.

Error: The client (role) does not meet the signature constraints for the service (method) that he is trying to use in the given resource.

Time Constraints Check

Time constraints restrict the access of methods by clients (roles) in the system to a limited time period.

Error: The client (role) does not meet the time constraints for the service (method) that he is trying to use in the given resource.

We cite here an example to explain this process.

Consider that the role CSEDeptHead is assigned to user Ting, the role CSEFaculty is

assigned to users Ting and Steve and the role CSEUndergrad is assigned to user Mumtaz. Now consider three methods, addCourse(), updateCourseCapacity() and registerCourse() present in Course Database resource. If the security officer grants a *positive privilege* to the roles, CSEDeptHead and CSEFaculty, to use addCourse() then Ting may use this method if he signs in as either CSEDeptHead or CSEFaculty. Next, assume that updateCourseCapacity() takes as parameter an integer that represents the enrollment capacity for a course and that both CSEDeptHead and CSEFaculty have a positive privilege to use this method. A constraint is put on this parameter such that CSEDeptHead may modify the enrollment capacity up to 40 whereas CSEFaculty may do so only up to 30. Now Ting, as the CSEDeptHead may be able to update the enrollment capacity of a course to 35; however, as a CSEFaculty he will not be able to update the enrollment capacity to 35 (*signature constraints*). Next, suppose that a constraint is put on the usage of registerCourse() method, which role CSEUndergrad has a positive privilege to, such that this role can access this method only between Jan 1, 2001 and Feb 28, 2001. Now user Mumtaz will have a *time constraint* on the usage of this method and she cannot access this method before or after this time period.

4. Implementation of Time Constraints and Signature Constraints

Time Constraints and Signature Constraints are two major changes we made to USR this semester. Time Constraints are used to limit the time period when a method can be invoked. Signature Constraints are used to limit the values (return and parameter) under which a method can be invoked. Signature Constraints are different from database integrity constraints. The limitation of Signature Constraints is based on the role, while the limitation of database integrity constraints are for all the roles.

4.1 Time Constraints

In a lot of situations, we need to limit the time when the client can access the system. In the system last semester, the system assumed that resource/service/method and role/user were valid only for a certain period of time. We found it not reflecting the actual situation and inflexible when we investigated the security logic this semester. In real life,

the resources/services/methods themselves do not expire, instead they just become inaccessible for certain roles. It is, in fact, the definition of user roles' access to them that expires after the valid time period. The same logic also applies to the definitions of user and role. It is the assignment of a role to a user that should expire instead of a user or a role itself. This semester, we made modifications to the existing system to support this more reasonable logic.

In the database schema, we added two more optional time-related columns to tables that store the relationship, including user_role, role_resource, role_service, role_method and role_ip. These two columns will store the valid time interval for the relevant access if there is any. For example in the role_method table, each tuple now becomes [ur_id, r_id, m_id, begin_date, end_date] instead of the previous simplistic tuple of [ur_id, r_id, m_id]. In the security server, we changed our client (user) registration logic and constraint-based authorization logic accordingly. These logic will check if the access is in the valid time period if there are any time constraints for the access. In both Security Policy Client and Security Authorization Client, we added relevant fields for the security officer to set up time constraints. In Figure 7, we give examples of the changed Security Policy Client and Security Authorization Client.

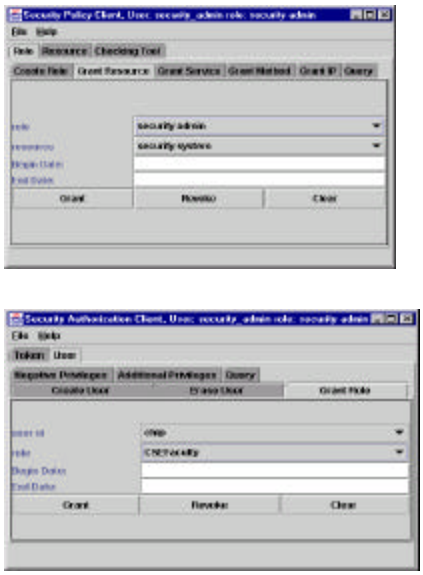


Figure 7: Security Client with Time Constraint Support

4.2 Signature Constraints

Signature Constraints basically are to extend role-based method invocation based on allowable data values (both return and parameters). It is very critical since they will allow us to limit the conditions under which a method of a service for a resource may be invoked based on role. Using Signature Constraints, multiple users can be allowed to access the same method but in different ways based their roles. Take the following scenario for an example: in the course database system, both faculty role and student role should be able to query available courses. But in some transition period, the student role should only have an access to the classes in the current semester, while the faculty can have an access to classes from both the current semester and the next semester for reviewing purposes. It is the Signature Constraints that take care of this. When getClass() method in the course database system is invoked, it will take the semester name as a parameter. The security officer can add a signature constraint in Security Policy Client for this method to limit the value of the semester name for these different roles. For the faculty role, they can query classes from both the semesters, while the student role can query classes only from the current semester.

In order to support the signature constraints, we changed the database, security server, and policy client.

1. Database Changes: We added two more tables, signature and role_signature. We required the resource server to publish and register its method signature information, including the number of parameters each method is taking, the data type of each parameter and its description. This information will be stored in signature table. The role_signature table is used to store the assignment of signature constraints to the role. In order to support multiple parameters and future logic relationships of the constraints to these parameters (AND, OR, NOT, etc) [3], we use a preformatted string to keep the signature constraints for each method. The string is in the following format:

   NOT **[parameter name, parameter data type, (parameter constraint)]** ***AND/OR*** **…….**

   The italicized strings are optional. The first two parts in the bracket are easy to

[3] As the first prototype of signature constraints, the current system only supports AND

understand. The third part will change according to the data type. Each data type needs to follow a rule. Now we accept three different data types. For STRING, the parameter constraint should be an enumeration of accepted strings separated by comma. For BOOLEAN, it should be True or False or both of them separated by comma. For NUMBER, it should follow *"min-max" (for example, 0-40)* for one range or one number for a specific value *(for example, 30)*. Multiple ranges and values will be separated by commas.

For example, in course database system, the method updateClassCapacity() takes two parameters, one is the course number and the other is the new capacity. We now add a constraint so the faculty role can only update the capacity of the class CSE123 and CSE124 up to 30. This constraint string will be as follows: **[course number, STRING, (CSE123, CSE124)] AND [capacity, NUMBER, (0-30)].**

2. Security Server Changes: We added several new methods to support signature constraints: **signatureRegister()** for the resource to register its signature information for each method, **grantMethodSignature()** for assigning signature constraints to a role, **revokeMethodSignature()** for revoking the signature constraints from a role and **getMethodSignature** for Security Policy Client to get the signature information for the method. A method, **changeToConstraintString** will be provided to help the resource server to change the method call to a preformatted input string, **[parameter name, parameter type, parameter value]**. Two classes are added for parsing the constraint string and compare the constraint string in the database with the input string. The parser will change the constraint string and input string to two vectors and the comparator will match each parameter according to its data type and return a Boolean expression for the constraint-based authorization check. For example, when the course database client using faculty role is calling a method, updateCapacity by updateCapacity("CSE123", 35). The resource server will first call changeToConstraintString method to generate a string like **[Course Number, STRING, (CSE123)] AND [capacity, NUMBER, (35)].** This string will be a parameter for the constraint-based authorization check together with the role id, resource id and method id. There are constraints to both of two parameters. The constraint-based authorization check will then parse this string and the string retrieved from the database to compare each parameter. Since CSE 123 is within

(CSE 123, CSE 124), the first parameter check will return True. It continues to the second parameter. Since 35 is out of 0-30 range, this parameter check will return False. The constraint string specifies that these two constraints are connected by AND, so the comparator will return False to the constraint-based authorization check and reject the client's update to the class capacity.

3. Security Policy Client Changes: The security officer will use the Security Policy Client to add and remove signature constraints. To avoid possible illegal accesses, if the method has not been assigned to the role, the security officer cannot set the signature constraints. For different data types, GUI for assigning signature is different which makes the system user-friendlier. Security Policy Client will also generate the constraint string according to the security officer's input and return to the security server for putting it in the database. In Figure 8, we give examples of the changed Security Policy Client GUI.

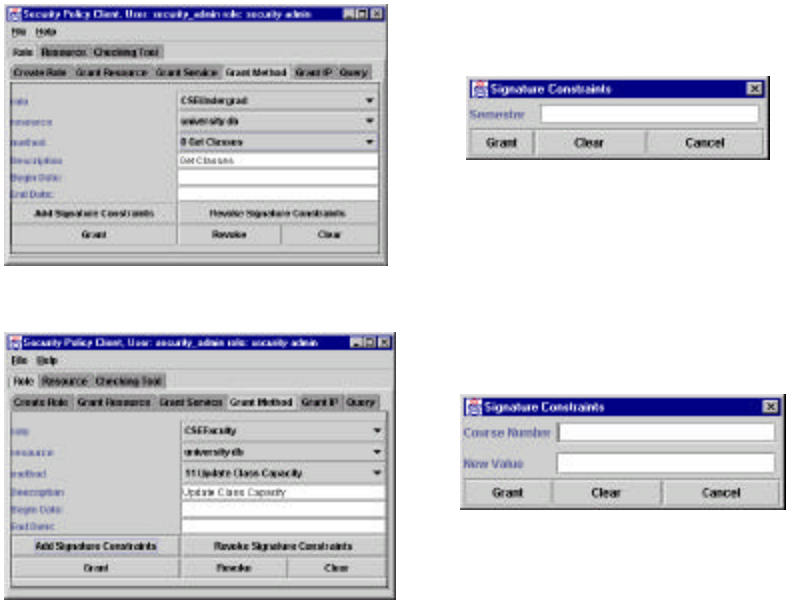


Figure 8: Security Policy Client with support for Signature Constraint

5. Implementation of Tracking and Analysis Tools

5.1 Tracking Tool

Any security system cannot be called perfectly secure unless all the calls made to the resources in the security system can be recorded and traced for any security violations. To allow this facility in our system, we have developed a tracking tool that will dynamically and automatically start recording relevant information about every attempt that is made to access the resources by any clients. This tool will, therefore, let security officers monitor all the activities that occur in the environment, right from the time a client starts a session with the security system till the time he logs out of the system. This tool has been added to USR and the security officers can use this tool to generate a list that contains all such information from a specific start date to the current date.

5.1.2 Implementation of Tracking Tool

Since almost all of the security checks occur in the hasClientRight() method, the major part of the implementation of tracking tool has been incorporated within the hasClientRight() method. Every time a client requests any kind of access to the system resources, the checks described in section 3.1 or section 3.2 are performed, which essentially pass through the hasClientRight() method. All the key information, which is either passed through the parameters of the method called (example, token), or is generated by the system itself (example, time and date), or which reflects the status of the access (success or error) etc. is recorded. Precisely, the data recorded comprise of the following:

1. The date and time when the user tried to access the service
2. The IP address of the machine from where the user signed in
3. The user id of the client
4. The role id of the user

5. The token assigned to the user if the authentication check was successful (if the authentication step was unsuccessful then the token is supposed to be '0')
6. The resource id that the user was accessing
7. The service (or method id) that the user was trying to use
8. The status of the access viz., SUCCESS or ERROR
9. A precise description of the status (why an error was caused, etc)

A table called log is maintained in the database, which keeps all these data. The information can then be tracked at any time by the security officers through the security policy client.

Figure 9 shows the changed Security Policy Client to support the tracking tool and also gives an example of a log file generated through the security policy client.

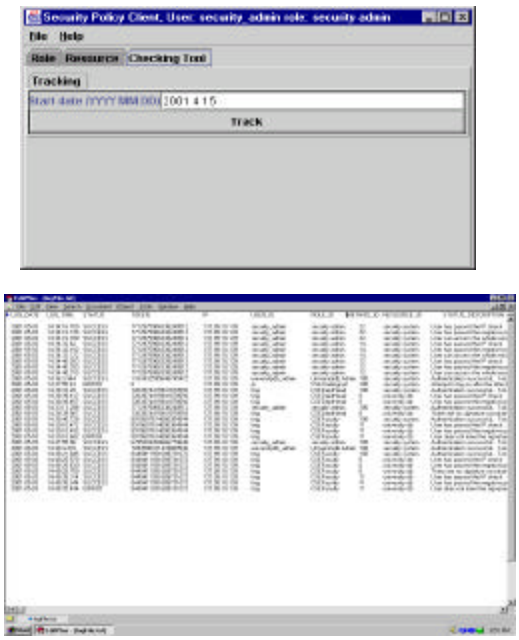


Figure 9: Security Policy Client GUI and Output of Tracking Tool

5.2 Analysis Tool

Static Analysis Tool is an attempt to add introspection capabilities, which augment the definitional capabilities with analyses, to the Role-Based security system prototype. The current approach to analyze the actual source code for each of the Java

resources that are developed, allows tracking not only the method on a resource that has been directly authorized to a role, but also the other resources and/or methods that are called to realize a particular method. Clearly, a significant assumption regarding the availability of source code was made to conduct analysis. Such code will not typically be available for a legacy or COTS application. The developed Static Analysis Tool is the initial version of introspection and analysis capabilities.

The Static Analysis Tool analyzes a class, which implements all the methods of a given resource. It inspects all the method definitions one after the other to find any other method called inside the one under inspection. When it finds such situation, it does the appropriate database queries to retrieve the Roles to which these methods are assigned. It creates an alert message for each such finding in the result file. The result whether the Authorized Roles for one method match with those for the other are reported back in the result file.

5.2.1 Implementation of Analysis Tool

The Static Analysis Tool inspects a given Java class in the following manner:

1. All the declared methods for that class are obtained.
2. Each method definition in the source code for that class is checked to locate any call to the other method(s) of that class.
3. When located, an entry of alert message is recorded in the result table.
4. Appropriate database queries are done to acquire the knowledge of the Roles to which the methods are assigned.
5. The resultant Authorized Role sets of the outer and inner method are matched against each other.
6. The result is stored in the alert table in the output file.
7. Every time when the tool runs, it stores the results of the previous analysis in the file 'Static_Analysis_Result_Old.txt" and stores the new results in the file "Static_Analysis_Result_Current.txt".

As the tool uses its own authorization for the security manager, and is implemented with Java classes it is capable of analyzing the Java classes defined for any resource of the current prototype.

Following is a list of expected results obtained from Static Analysis:

- **NO Roles Found.....** : No Roles were defined for both the methods.
- **Roles MATCH Exactly** : The Roles assigned for both the methods match exactly.
- **---- MISMATCH ALERT ----** : The outer or inner method have no Roles defined for it, while the other one is having some Roles defined for it. This situation must be inspected to find the possible security violation.
- **---- MISMATCH ALERT (# of Roles) ----** : The Roles assigned to the outer method form a subset of the Roles defined for the inner method. This situation must be inspected to find the possible security violation.
- ****** Roles MISMATCH ****** : The Roles assigned to the inner method form a subset of the Roles defined for the outer method. This is clearly the security violation, since the role that is not authorized for the inner method can call the same via outer method, for which it is authorized.

Hence, the tool creates different types of alert messages according to the result of Role matching for both the inner and outer method. The security officer should observe and take a necessary step for each one of them.
In case of any “MISMATCH ALERT”, the security officer should inspect the identifiers and Role sets for both the methods. If a private method of that class, which is not assigned to any of the Roles in the system is calling a public method assigned for some Roles, then that might not be a potential security violation unless that outer private method is called inside another public method. In such a case if the Roles of that third method do not match with those of the inner method, then that would prove as a security threat.

Similarly, if a public method assigned to certain Roles is calling a private method may not be a security violation unless the inner method calls a third public method. Here again if the Roles for that third method do not match with those of the inner method, then that would be a security violation.

The "MISMATCH ALERT (for # of Roles)" also should be inspected by security officer to see whether the Roles for the outer method form a proper subset of the Roles for the inner method.

The case of "Roles MISMATCH" should be avoided in any case as that tells the clear mismatch in the Roles for the inner and outer methods. This can be demonstrated with an example below.

E.g. We analyzed "UniversityDBImpl", which is the implementation of the methods of the resource "UniversityDB". The analysis of this example can be seen in Figure 10.

public boolean registerCourse(....)
{
:
addCourse(token, course);
:
}
	♦ Roles for registerCourse() : CSEUndergrad, ta
	♦ Roles for addCourse() : CSEDeptHead
	
	Tool reports
	"**** Roles MISMATCH ****"
	
	This is indeed a security violation !!

public boolean registerCourse(....)
{
:
getClassDescription(course,isCoursenum);
:
}
	♦ Roles for registerCourse() : CSEUndergrad, ta
	♦ Roles for getClassDescription () : CSEDeptHead, CSEFaculty, CSEUndergrad, ta
	
	Tool reports
	"---- MISMATCH ALERT (# of Roles) ----"
	
	No potential security violation here.

Figure 10: Output of Analysis Tool in different scenarios

Figure 11 is the screen shot for analysis client while the various alert messages for different situations are depicted in Figure 12.

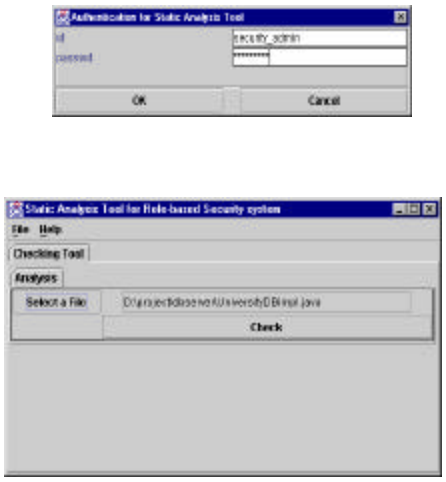


Figure 11: Static Analysis Tool for Role-based Security System

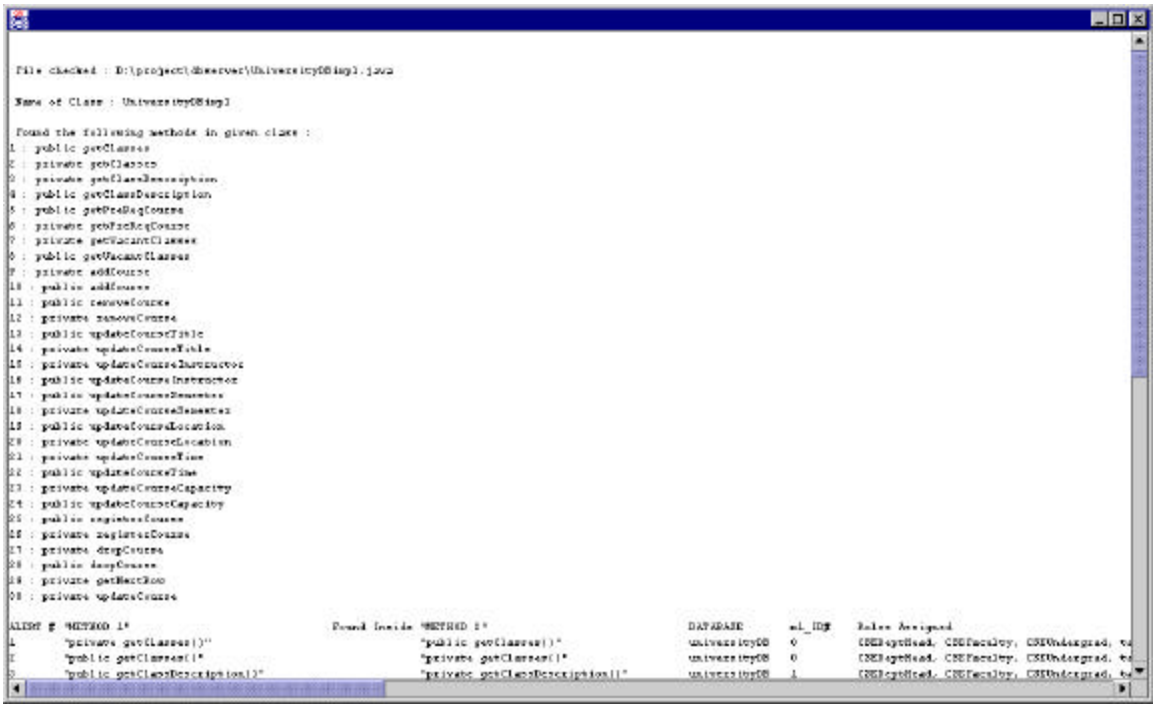


Figure 12: Output of Static Analysis Tool for Role-based Security System

6. Conclusion and Proposed Future Work

6.1 Proposed Future Work

1. Mandatory Access Control (MAC) and Role-Based Access Control (RBAC). It will be a good concept to make a system that combines the concepts of both RBAC and MAC.

   (*We feel that the best that can be done is to have a Role-based Model with MAC feature. This would require changing our database schema to account for classifications of database resources and security clearance levels of users.*)
2. Cryptosystems. There are several public and symmetric key cryptographic systems in use that could be adapted to our security system. This is something that would need to be done in order to have a credible security system.
3. JINI/CORBA bridge. The clients should be able to use resources/services/methods registered with CORBA as well as JINI i.e. there should be a facility to share resources/services/methods regardless of the middleware they are registered with.

   (*Slightly complicated. JINI resource needs to add CORBA support and CORBA resource needs to add JINI support. There should be a way to coordinate this.*)
4. Common Use Resource and Global Clock. A common use resource is a resource shared within a computing environment for whatever the reason. JINI and CORBA should be able to share a common resource. A Global Clock is a common resource used to make sure that database entries are entered in the proper sequence in a distributed environment. Implementing a Global Clock as a common use resource is an important improvement to our implementation.

   (*Kind of complicated. A brand new resource is needed and we should consider its interaction with security server and other resources.*)
5. Single Security Server. There should be only a single security server for all the resources regardless of the middleware.

   (*Very easy to change our current more sophisticated security server to CORBA environment. The CORBA wrapper should be kept and the security server and client implementations should be replaced.*)
6. (Resource, service, method) tuple. The (resource, method) pair should be changed to

a (resource, service, method) tuple in order to identify each method. In the current version, the service is ignored.

(*Not difficult, but need to be careful. The codes for supporting service is ready in the security server and clients, extensive tests and debugging are needed for finally putting it to use.*)

7. OR and NOT Signature Constraints. The relation assumed amongst the signature constraints currently is that of AND. Future work should allow OR and NOT relations as well.

   (*Easy to do. When we design the signature constraint implementation, we already consider this. Small changes to parser and comparator classes will be required.*)

8. Return Type Constraints. Currently, only the arguments or parameters passed into methods are being taken into consideration while defining signature constraints. In the future, however, the return value must also be considered while defining signature constraints.

   (*Easy to do. Changes in the database for storing return parameter and the constraints on return parameter are needed.*)

9. Expanding data types. The data types currently allowed for signature constraints are integer, boolean and String. In future, a constraint on all other remaining primitive data types (like float, long, etc) as well as more complicated data types like objects should be allowed.

   (*Support for primitive data types is simple to implement, but for more complicated data type, it is difficult. We haven't figured out a way to do this.*)

10. Integrity Constraints. Set up integrity constraints on all elements of the resource database. The integrity constraints are the limits to all Signature Constraints and will ensure database integrity.

    (*Very easy to do, and would be a good complement to the security model*.)

11. Groups. The concept of group refers to several users coming together to use the same resources. An example would be a team of advisors for a single student.

    (*Not difficult. But need to clarify concepts before implementation.*)

12. Constraints on Users. Currently, we only concern ourselves with the constraints based on role. However, there should be some way to assign constraints to an individual.

    (*Very easy to implement. It can be done in the same way as negative privileges are assigned to users.*)

13. Tracking Tool. In the Tracking Tool, currently a log file is returned based on a start date from where the data should be tracked. A future version should allow information to be tracked based on any of the 9 different types of information recorded (excluding perhaps, the status description).

    (*Very easy to implement. The required additions should be made to the logFile() method in SecurityOfficerImpl.java with the corresponding changes also made to the Tracking tool GUI.*)

14. Analysis Tool. Currently, the Static Analysis Tool is capable of inspecting Authorized Roles for a given method of a resource. A future enhancement should also allow analysis of combinations of methods assigned to a Role, such that, when a Role is authorized for a given set of methods, the tool should be able to test whether that set is necessary and sufficient, and whether it causes any kind of security violation.

    (*Easy to do. Change the GUI interface to include tabs & drop-down menus for "Select a Resource" & "Select a Role" in the Tool window to select a role from existing roles for a given resource. Query the "role_method" table depending upon the role selected. Then, make use of FileAnalysis and MatchRoleMethod classes to analyse the methods assigned to the Role selected.*)

15. Analyzing multiple source files. The Static Analysis Tool inspects a single source file, which defines a given Java class implementing the resource methods. It should be altered to check a set of classes, which collectively realize that resource.

    (*Slightly complicated. An extensive use of java reflection package will be required. An alternative could be to use compiler tools like Lex and Yacc, which are tools used to generate lexical analyzers and parsers, to analyze the source code.*)

16. Role Deconfliction. This would be a function of the analysis tool where certain roles cannot be present if other roles are present.

    (*This will require a good understanding of the Chinese Wall Problem. For example, a budget officer should not also be a procurement officer because the budget officer would fund everything that he or she procures. There is a conflict of interest. The conflicts would most likely be established in policy, so we would need a way to collect and then check for these policy conflicts. We might want to call this Policy Constraint Management.*)

17. Agents. Our current security model cannot handle agents. A good feature of the system would be that it integrates our security prototype with an agent environment.

18. Date Selection GUI. Scroll-down menus should be used for inputting dates rather than letting the user input the dates through text box (as is done in the current version) so as to prevent the user from making any mistakes in the acceptable date formats.
(*Easy to do. Only GUI changes are needed, but need to make sure every place has been changed and throughout the system, the date format should be consistent.*)
19. Role Assignment GUI. The system should incorporate an FTP like double window exchange configuration to make it easier for the security officer to select privileges for a user role or roles for a user.
20. Solution to JINI's problems. JINI development team should be contacted in order to resolve JINI's memory relocation problems on NT machine.

6.2 Conclusion

This semester, we successfully extended the JINI security system. The major changes we made to the system include:

1. Rewriting the GUI's of all the clients to add drop-down menus at almost all places where user input is required. In addition, the selections in the drop-down menus are made to be always consistent with the most up-to-date information in the database. The system is thus made user-friendlier, easy to use now and foolproof.
2. Designing and implementing the signature constraints concept from the very beginning and successfully proving the prototype.
3. Redesigning and implementing the time constraints for the security system.
4. Addition of Dynamic Tracking and Static Analysis tools to the security system.
5. Fixing the bugs in the older system. The current version of our system is much more unbreakable and robust as compared to the older version.
6. Designing and implementing Additional Privilege for the security system. The codes for both security server and clients are ready to use though the concept was discarded later.

The user manual and UML documentation for the system are included in the Appendice

6.3 Work Breakdown by Student

We had a great team of three members this semester. The work breakdown is as follows:

- **Qi Jin (Qi)**

  Qi continued her work from last semester. She helped other group members to become familiar with the security model and the previous security system. She was also responsible for fixing bugs in the previous system. She designed and implemented Time Constraints. She was the primary designer of Signature Constraints. Qi rewrote GUI interfaces for the previous system (both security system and course database system) and added new GUI interfaces to support new Time Constraints and Signature Constraints features. In addition to this, she also designed and implemented Additional Privilege before it was removed from the security model.

- **Mumtaz Lohawala (Mumtaz)**

  Mumtaz was responsible for designing and implementing the Tracking Tool in the security system. She contributed to the implementation of Signature Constraints by writing the major parser and comparator classes. She assisted in writing the GUI interface for the Analysis Tool. Mumtaz also modified the Authentication Client GUIs to conceptually change the log-in process.

- **Nayana A Limaye**

  Nayana was the designer of the Static Analysis Tool for the existing Role-based security system. She also implemented the Static Analysis Tool to function independently from JINI or CORBA. She added to the Tool, the Authorization of its own to make it portable to any security system. She suggested some additions to database schema, which Qi and Mumtaz implemented. These additions would be helpful for the future version of Static Analysis Tool.

Acknowledgement

We would sincerely like to extend our vote of thanks to Prof. S. Demurjian and Charles. E. Phillips for all their guidance and help throughout the course of this project.

Appendix A

User Manual to run the JINI Security System

Software Requirements

1. JINI as the lookup service
2. Oracle for database
3. Java 2 Platform SE v1.3 or greater

Required Directory and Files

When you distribute the system on multiple machines, first you have to change the corresponding batch files to make sure all the servers and clients can find JINI. Here is a guide to put source/class files.

Security Server: common/, server/ and startserver.bat

Security Policy Client: common/, policy/ and startpolicy.bat

Security Authorization Client: common/, enforce/ and startenforce.bat

Course Database Server: common/, dbserver/ and startdb.bat

Course Database Client: common/, dbclient/ and startdbclient.bat

Static Analysis Tool: analysis/, dbserver/, server/ and starttool.bat

Steps to start the system

Note:

[1] One always needs to have JINI in the environment.

[2] One always needs to run the Security Server in order to run any of the security clients.

[3] One always needs to run both Security Server and Course DB Server to run Course DB client.

[4] The static analysis tool does not require any other system components running.

Here are the steps to running the demo on the NT machine:

1. ***START JINI***

   From a DOS Command Prompt window:

   a. C:\> cd files

   b. C:\files> startgui

      This will give you a GUI for JINI, from this window: go to FILE, then double click (open) jini_win32_test.property, which will give you a new window called " START SERVICE", do the following:

      i. In the "Reggie" Tab, check the following:

         Codebase: make sure "hostname" is changed to "localhost" or IP address of this host

         Log Directory: C:\tmp\reggie_log### (must fill in a new number for ### or delete the current number from “C:\tmp” directory)

      ii. In “Run” tab, click

         - Start RMID
         - Start Web Server
         - Start Reggie
         - If you see the screen like Figure A.1.1, JINI is up.

2. ***START Security Server***

   From a second DOS Command Prompt window:

   a. D:\project> startserver

   b. If you see the screen like figure A.1.2, the security server is up and registered with JINI.

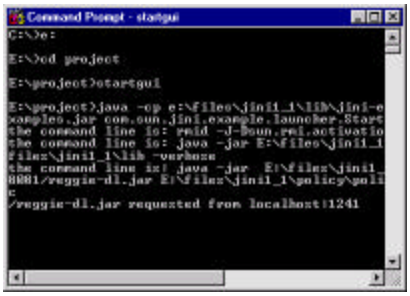


Figure A.1.1 JINI is up

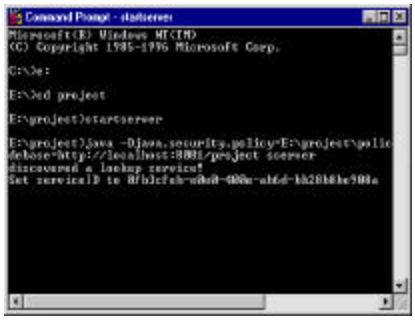


Figure A.1.2 Security Server is up

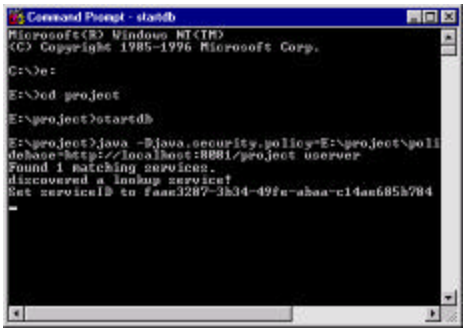


Figure A.1.3 Course Database Server is up

Figure A.1 DOS windows for starting JINI and servers

3. *START Policy Client*

From another DOS Command Prompt window:

a. D:\project> startpolicy

"Security Policy Client Authentication" frame will pop up. Input user name and password first (default user: security_admin and default Passwd: chongsong). Click on "Select Role", it will bring a list of roles. Select one of the roles and click OK. The Policy Client will pop up. See Figure A.2 for Authentication frame and Security Policy Client.

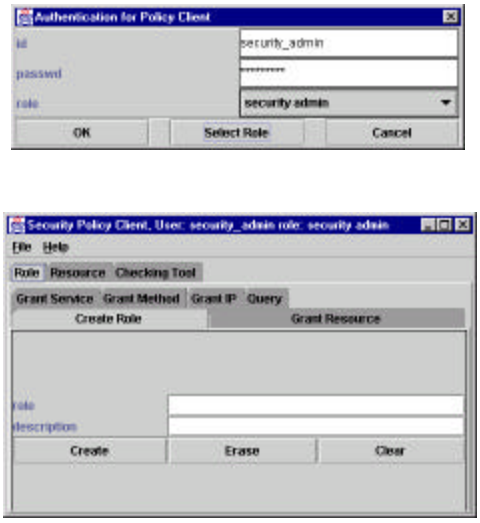


Figure A.2: Security Policy Client Authentication and GUI

Figure A.2: Security Policy Client Authentication and GUI

4. *START Authorization Client*

From another DOS Command Prompt window:

a. D:\project> startenforce

"Security Authorization Client Authentication" frame will pop up. Input user name and password first (default user: security_admin and default Passwd: chongsong). Click on "Select Role", it will bring a list of roles. Select one of the roles and click OK. Security Authorization Client will pop up. See Figure A.3 for Authentication frame and Security Authorization Client.

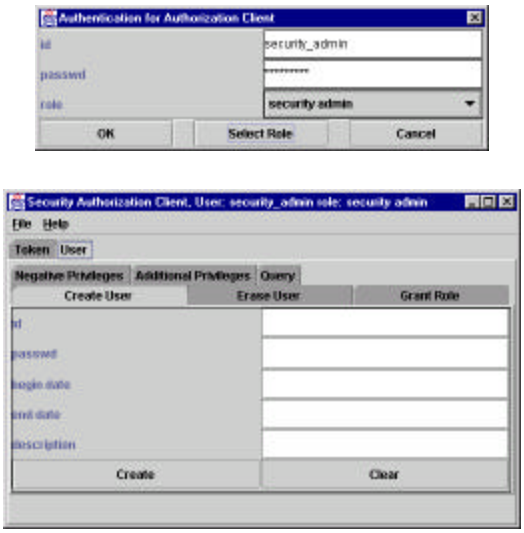


Figure A.3: Security Authorization Client Authentication and GUI

Figure A.3: Security Authorization Client Authentication and GUI

5. ***START Course Database Server***

From another DOS Command Prompt window:

a. D:\project> startdb

b. If you see the screen like Figure A.1.3, the Course Database server is up and registered with JINI.

6. ***START Course Database Client***

From another DOS Command Prompt window:

a. D:\project> startdbclient

"Course Database Client Authentication" frame will pop up. Input user name and password first (see existing users and passwords). Click on "Select Role", it will bring a list of roles. Select one of the roles and click OK. Course Database Client will pop up. See Figure A.4 for Authentication frame and Course

Database client.

A list of a few existing users, and their corresponding passwords and roles for the universitydb client is as below:

- ✧ User: universitydb_admin, Password: security, Role: universitydb admin
- ✧ User: steve, Password: steve, Role: CSEFaculty
- ✧ User: ting, Password: ting, Role: CSEDeptHead, CSEFaculty
- ✧ User: chip, Password: chip, Role: CSEUndergrad

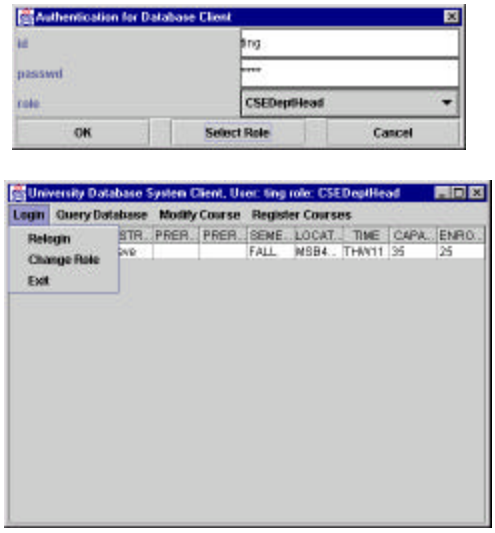


Figure A.4: Course Database Client Authentication and GUI

Figure A.4: Course Database Client Authentication and GUI

7. *START Tracking Tool*

Tracking tool is part of Security Policy Client. Start Security Policy Client as above. Select “Checking Tool” tab. Enter a valid date and click “Track”. It will prompt you to go to “logfile.txt” for log information. See Figure A.5.3 for Tracking Tool tab.

8. *START Static Analysis Tool*

From a DOS Command Prompt window:

a. D:\project> starttool

"Authentication for Static Analysis Tool" dialog box will pop up. See Figure A.5.1.

Input user name and password. (User: security_admin and Password:chongsong)

Click OK. "Static Analysis Tool for Role Based Security System" will pop up.

Click on "Select a File" tab and select the source file for the class, which you want to analyze. The absolute path for the file will appear in a text window. Click "Check".

See Figure A.5.2 Analysis Client.

Figure A.5.1 Analysis Tool Authentication

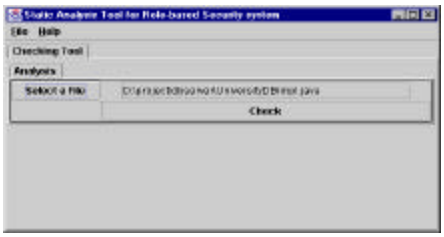


Figure A.5.2 Analysis Tool GUI

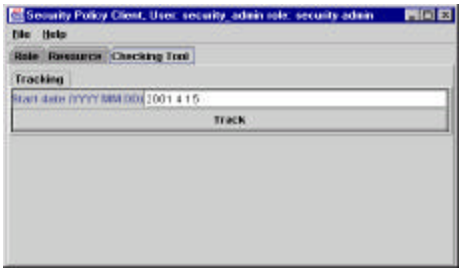


Figure A.5.3 Tracking Tool GUI

Figure A.5: Analysis Tool Client GUI and Tracking Tool Tab

Appendix B
ER and UML Class Diagrams for JINI Security System

Security System Database Schema

Figure B.1 is ER diagram for Security System database.

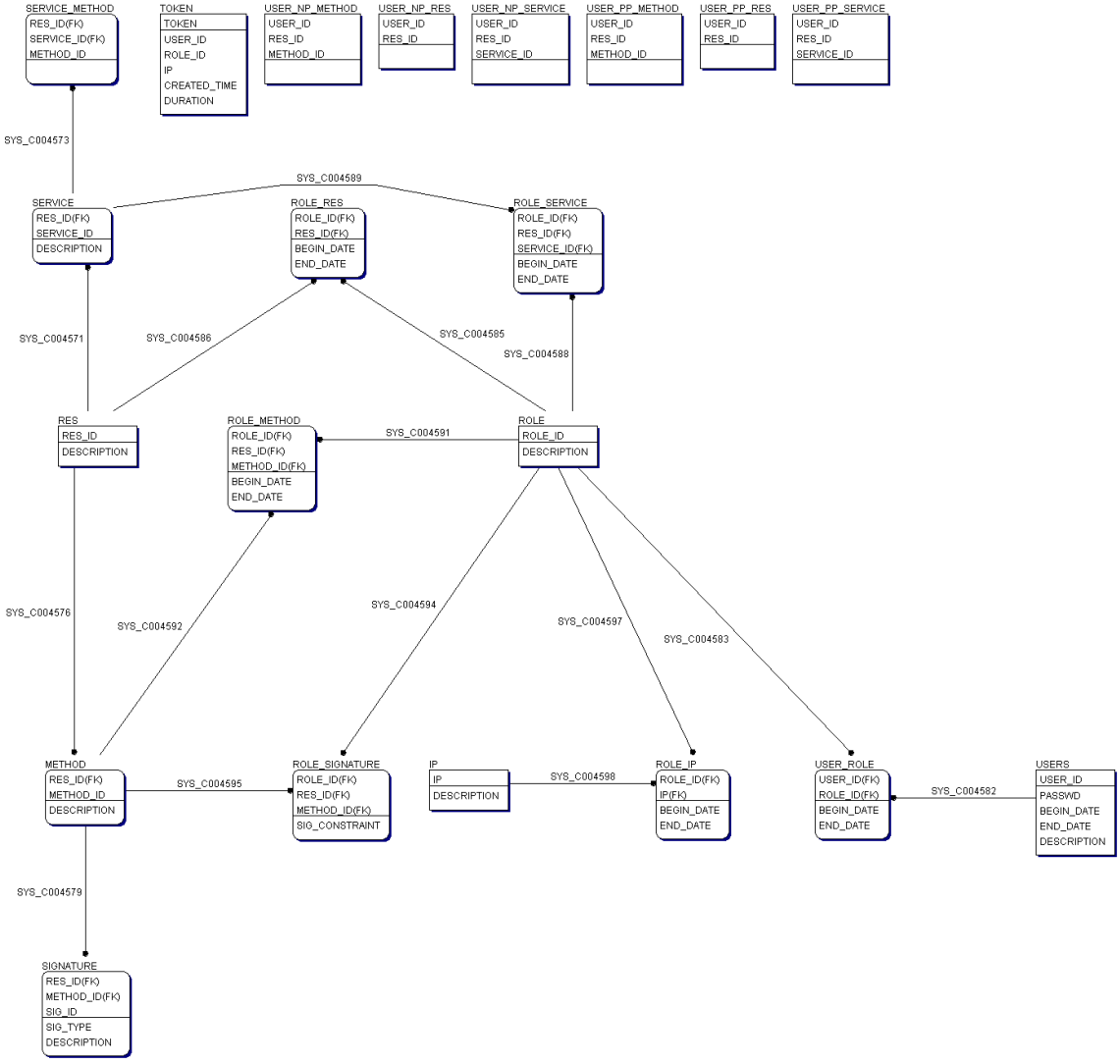


Figure B.1 ER diagram for Security System Database

Common classes (common package)

This package contains classes that are common to all the packages described below and hence have been placed in a separate package accessible to all. UML class diagrams for the classes in this package are in put in the other packages. The main classes and their descriptions are as follows:

1. *AuthenDialog()*: GUI interface used for authentication of any client/user.

2. *Const()*: Defines the constants which are basically the Oracle JDBC driver name; the URL, ID and password for accessing the Oracle databases for the security resource and the University resource; et. al.
3. *GuiUtil()*: Defines the utility static functions for all GUIs.
4. *SecurityInterface()*: This interface is used by all the resource clients and resource servers. The methods declared in this interface are implemented in the *SecurityOfficerImpl()* class.
5. *SecurityOfficerInterface()*: This interface is used by the security officer and the methods declared in this interface are implemented in the *SecurityOfficerImpl()* class.
6. *SecurityException()*: This is the exception class which treats every exception related to security.
7. *NoRightException()*: This is the exception class which is thrown when the client has no right.

Security server (server package)

Figure B.2 is the UML class diagram for Security Server. The main classes and their descriptions are as follows:

1. *sserver()*: It is a service class that publishes a proxy. It is a "wrapper" class that handles publishing the service item for the security server.
2. *SecuritySystemResourceID()*: This class defines all the constants which are required by other classes in the server package and also for filling out the method table when the security system resource (server) is up.
3. *SecurityOfficerImpl()*: It is a proxy object that will be downloaded by security clients. This class implements all the methods that are declared in the *SecurityOfficerInterface()* and *SecurityInterface()* interface.
4. *StringComparator()*: This class has methods that are used by Signature Constraints methods.
5. *StringParser()*:This class has methods that are used by Signature Constraints methods.

Figure B.2 UML Class Diagram for Security Server

Policy (policy package)

Figure B.3 is the UML diagram for Security Policy Client. The main classes in this package along with their descriptions are as follows:

1. *PolicyClient()*: This class is JINI wrapper for Security Policy Client.
2. *policy()*: GUI interface for creating policies regarding creating roles; granting resources, services, methods and IP's to roles; registering resources, services and methods; querying resources, IP etc.; tracking access information; et. al.

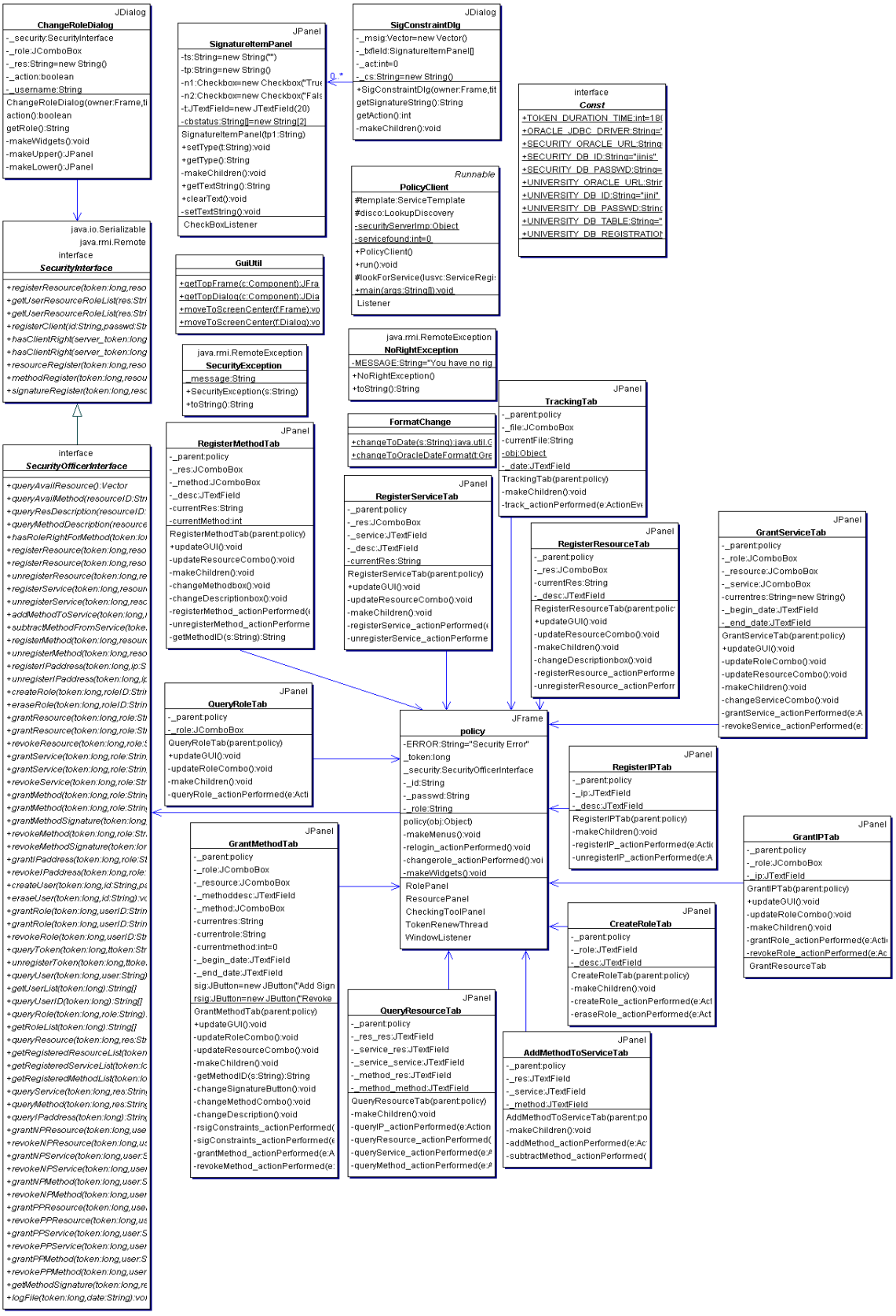


Figure B.3 UML class diagram for Security Policy Client

Enforce (enforce package)

Figure B.4 is the UML diagram for Security Authorization Client. The main classes in this package along with their descriptions are as follows:

1. *EnforceClient()*:This class is JINI wrapper for Security Authorization Client.
2. *enforce()*: GUI interface for enforcing the policy which involves creating and erasing users; granting roles, negative privileges and additional privileges to users; et. al.

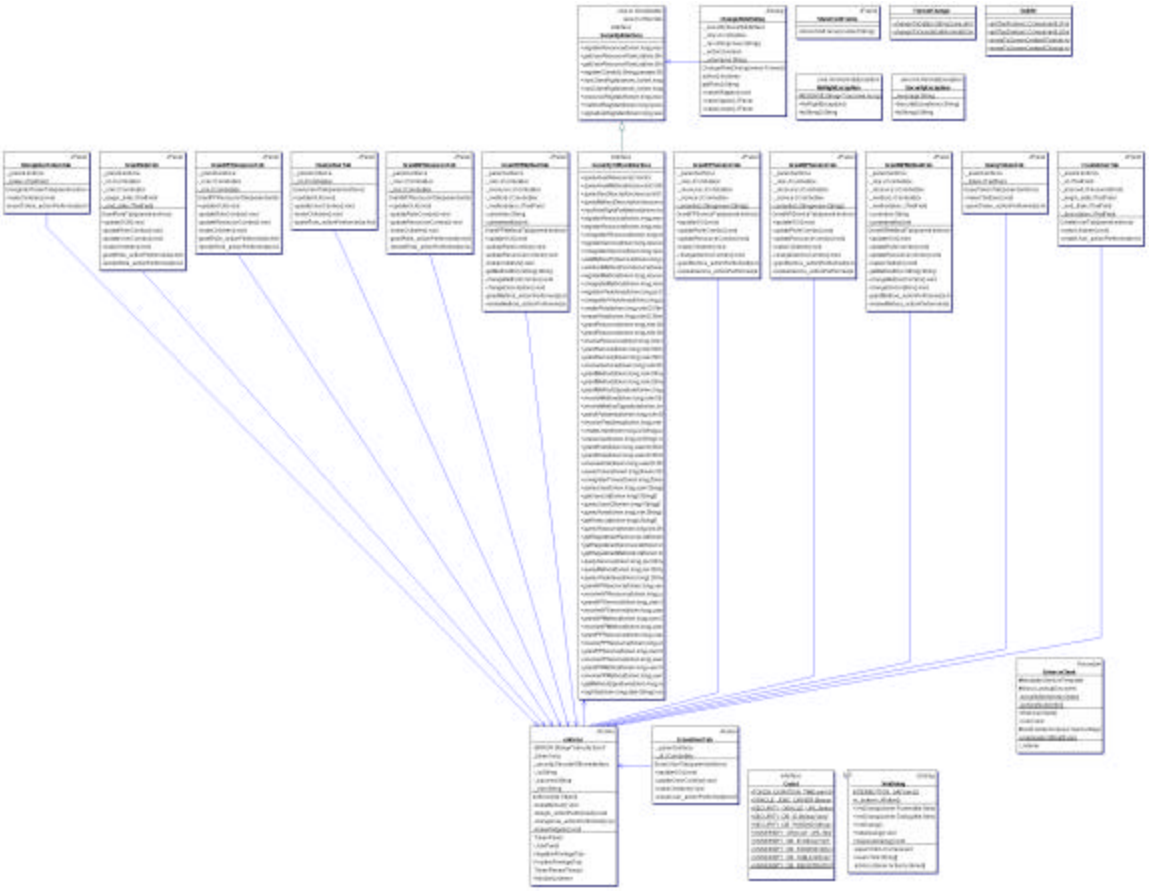


Figure B.4 UML class diagram for Security Authorization Client

Course DB server (dbserver package)

Figure B.5 is the UML diagram for Course DB server. The Course DB is a resource to which the security server will assign a unique resource id once it is registered. This package contains classes that implement the services offered by the Course DB resource. The main classes in this package along with their descriptions are as follows:

1. *userver()*: It is a service class that publishes a proxy. It is a "wrapper" class that handles publishing the service item for the University DB resource server.
2. *UniversityDBResourceID()*: This class defines all the constants which are required by other classes in the dbserver package. These constants are also used for filling out the resource/service/method/signature tables when the University DB resource server (dbserver) is up.
3. *UniversityDBInterface()*: It is an interface that declares the methods that are offered by the University DB resource.
4. *UniversityDBImpl()*: This class implements all the methods that are declared in the *UniversityDBInterface()* interface.

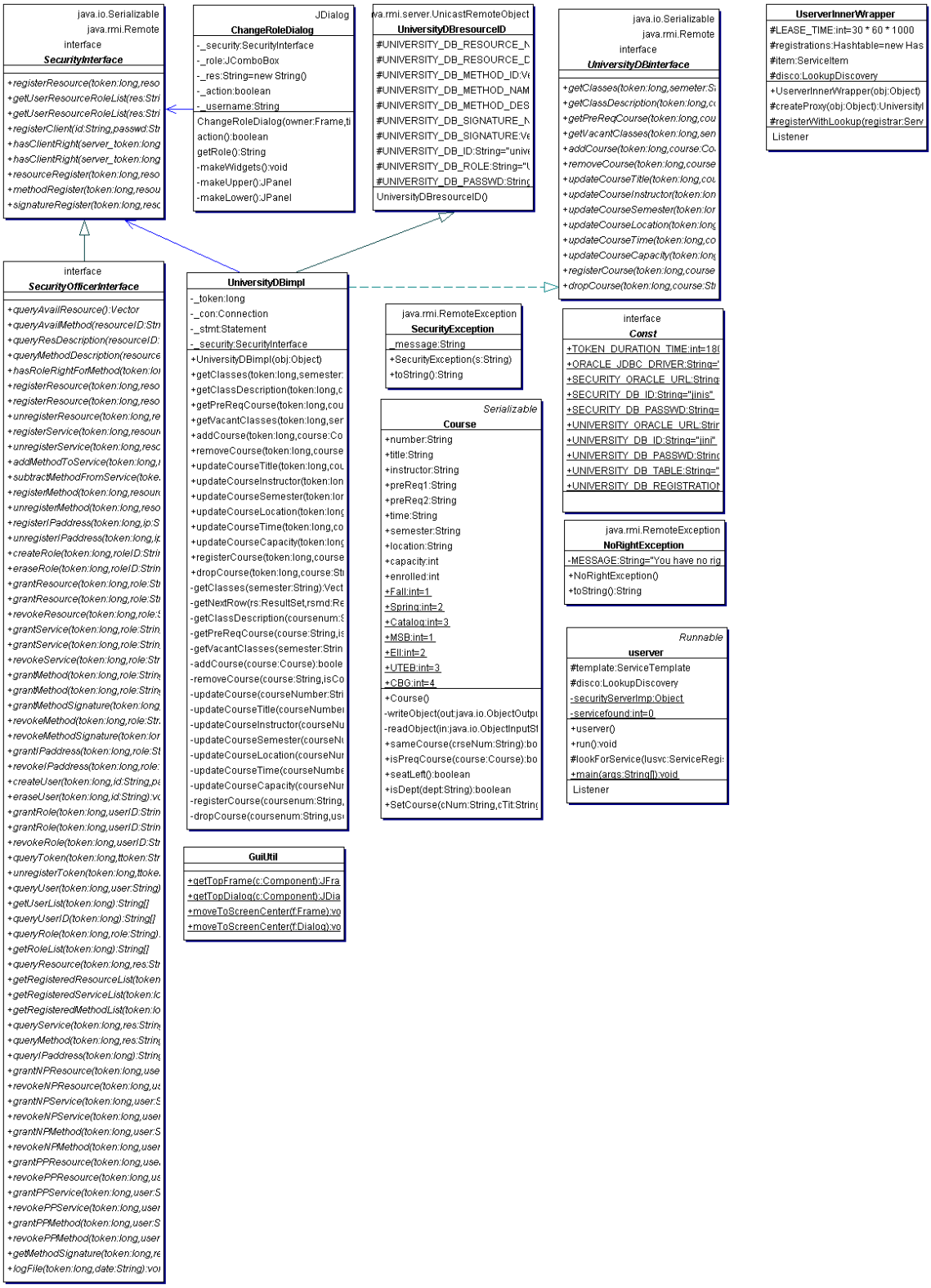


Figure B.5 UML class diagram for Course DB server

Course DB client (dbclient package)

Figure B.6 is the UML diagram for Course DB Client. The Course DB client is a client of the Course DB resource server. This package contains classes that are mainly the GUI interfaces for use by the clients to access the Course DB resource server. The main classes in this package along with their descriptions are as follows:

1. *GUIFrame()*: GUI interface for accessing the University DB server which involves querying the database for courses; registering and dropping courses; adding, modifying and removing courses; et. al.
2. *DBClient()*, *DBClientWrapper()*, *Dinner()*: JINI wrappers for Course Database Client.

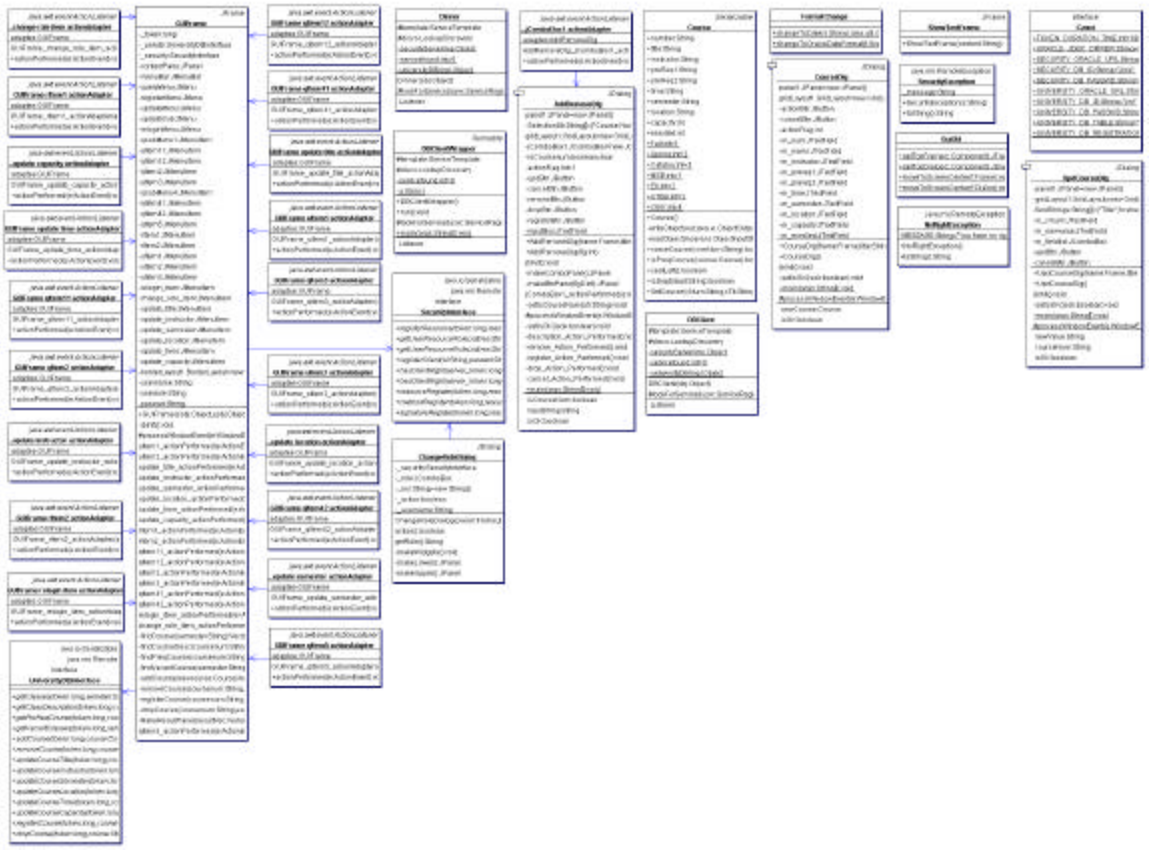


Figure B.6 UML class diagram for Course DB Client

Analysis (analysis package)

Figure B.7 is the UML diagram for Analysis Tool. This package has classes, which are used by the Static Analysis Tool. The main classes and their descriptions are as follows:

1. *AnalysisTool()*: The main GUI interface for selecting the file to be analyzed.
2. *MatchRoleMethod()*: Contains methods for performing analysis which involves matching the methods that a role can take; et. al.
3. *FileAnalysis()*: This class contains methods for fetching the inner method calls from the methods of a given file.

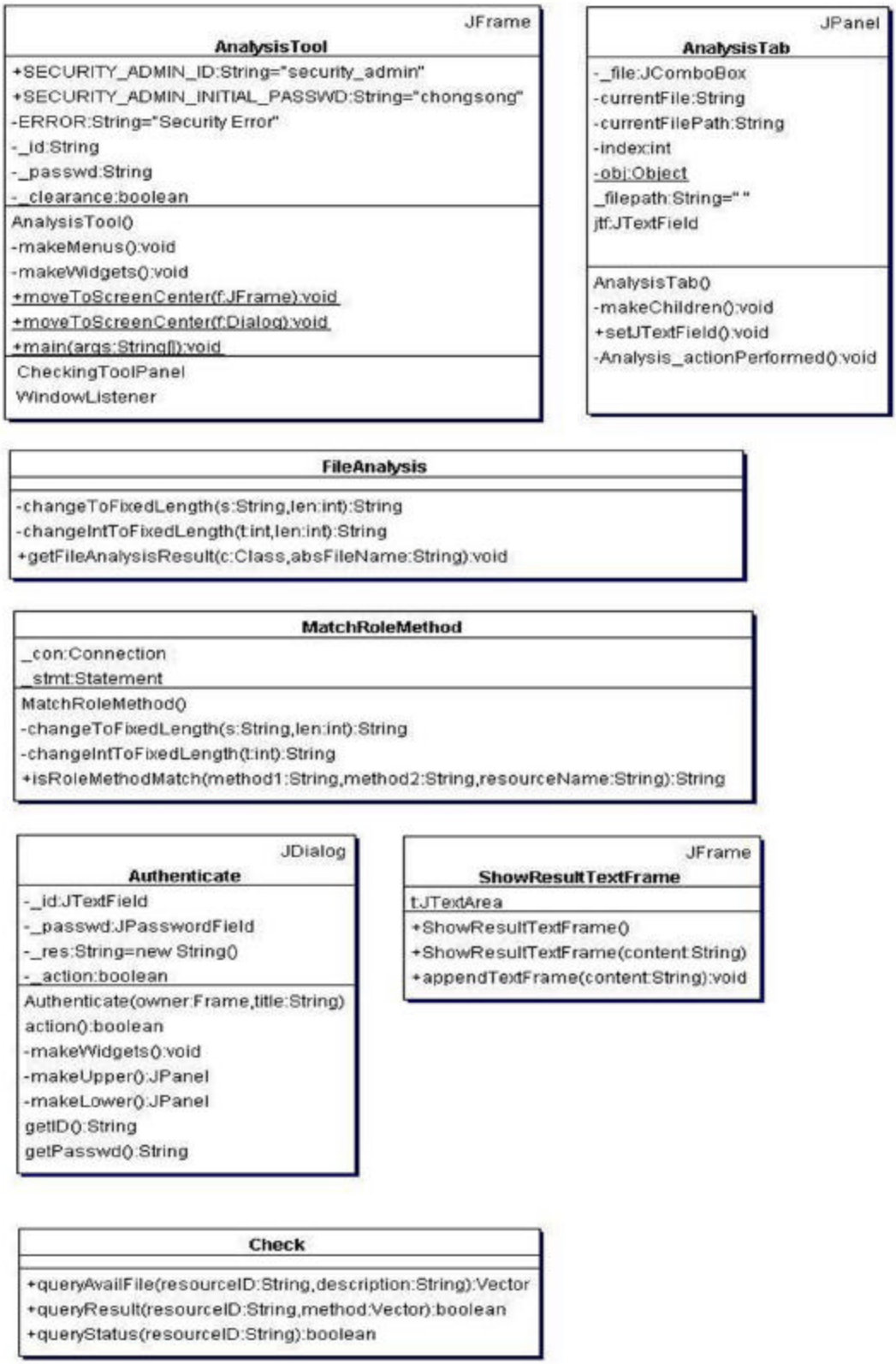


Figure B.7 UML class diagram for Analysis Tool