Resource Access Decision Facility 1

RAD facility is a mechanism for obtaining authorization decisions and administrating access decision policies. It enables a common way for an application to request and receive an authorization decision. The facility is intended to be used by security-aware applications.

This specification provides access decision functionality not supported by CORBAsecurity which is required in healthcare and other application environments. It is intended to be implementable using CORBAsecurity as a base; it is also intended to be implementable in ORB environments which do not provide CORBAsecurity. Athorization logic is encapsulated within an authorization facility that is external to the application. In order to perform an application-level access control, an application requests an authorization decision from such a facility and enforces that decision.

1.1 Conventions

1.1.1 Precondition/Postcondition semantics

In this specification we use a "design by contract" precondition/postcondition semantics. This means:

- An object which receives a call assumes that its preconditions are true at the time the call is received.
- If the preconditions are true, then:
- When the object returns from the call, the postcondition will be true, OR
- The object will throw an exception, indicating that it was unable to fulfill its contract.
- If ANY of the preconditions is not true, then the object's behavior is undefined, and, in particular, the postconditions are not guaranteed to be true when the object returns from the call.

1.1.2 Terminology

Access Decision Object (ADO) — The RAD object that implements access decision functions. From the perspective of a client requesting an access decision of RAD, this is the only interface that they are required to use. Although similar in function to the CORBAsec object of the same name, the RAD ADO has a different signature and semantics.

ADO Client — The immediate invoker of the RAD Access Decision Object. This could be an integral part of the application that controls the secured resources or it could be an interceptor that decides whether to allow the CORBA Request to reach the application.

Access Policy — The policy or rules that govern access to a secured resource.

AttributeList — A list of security attributes that are used to determine whether access should be allowed. An AttributeList may contain both static and dynamic attributes.

Authorization — The granting of authority, which includes the granting of access based on access rights.[1]

Component — A cohesive set of software services

Credentials — Information describing the security attributes (identity and/or privileges) of a user or other principal. Credentials are claimed through authentication or delegation and used by access control.1 The attributes contained in an RAD AttributeList are derived from CORBAsec credentials, if possible.

Dynamic attribute — A security attribute that can only be determined at the time an access decision is requested. Dynamic attributes are often based on the relationship between a principal and the secured resource (such as attending physician) and cannot be statically configured. This submission allows dynamic attributes to be resolved and used during the access decision computation.

Identity (attribute) — A security attribute with the property of uniqueness; no two principals' identities may be identical. Principals may have several different kinds of identities, each unique (for example, a principal may have both a unique audit identity and a unique access identity). Other security attributes (e.g. groups, roles, etc...) need not be unique.1

Naming Authority — Any organization that assigns names determines the scope of uniqueness of the names and takes the responsibility for making sure the names are unique within its name space. In the same way that ID values are meaningful only within the context of their ID Domains, names are unique only within the context of their naming authority.[2]

Operation — An action which may be performed on a secured resource (such as create, get, set, use..). Operations are represented within the RAD as strings.

Principal — A user or programmatic entity with the ability to use the resources of a system.1

1. This definition is taken from the OMG CORBAsecurity 1.2 specification. OMG document number ptc/98-02-01

2. This definition is taken from the OMG CORBAmed Person Identification Service (PIDS). OMG document number corbamed/98-02-29

Privilege (Attributes) Security attributes which need not have the property of uniqueness, and which thus may be shared by many users and other principals. Examples of privileges include groups, roles, and clearances. 1

Secured Resource A “secured resource” is any valuable asset of an application owner, which is accessed by an application on behalf of a principal using it, and access to which is to be controlled according to the owner’s interests.

Security attributes Characteristics of a principal which form the basis of the system’s policies governing that subject.[3]

Static Attribute A security attribute that is (typically) statically configured by an administrator. Examples would be access_id:john_doe or role:physician.

System An application or set of applications that interact with each other, interact with the RAD or implement RAD. System in this context is synonymous with application. Examples of systems might include a hospital or clinical information system, an ancillary system such as a lab or radiology system, or a financial/administrative system such as an ADT.

1.2 Facility Overview

1.2.1 Introduction

A simplified schema of application flow is depicted in Figure 1-1.

3.This definition is taken from the OMG CORBAsecurity 1.2 specification. OMG document number ptc/98-02-01

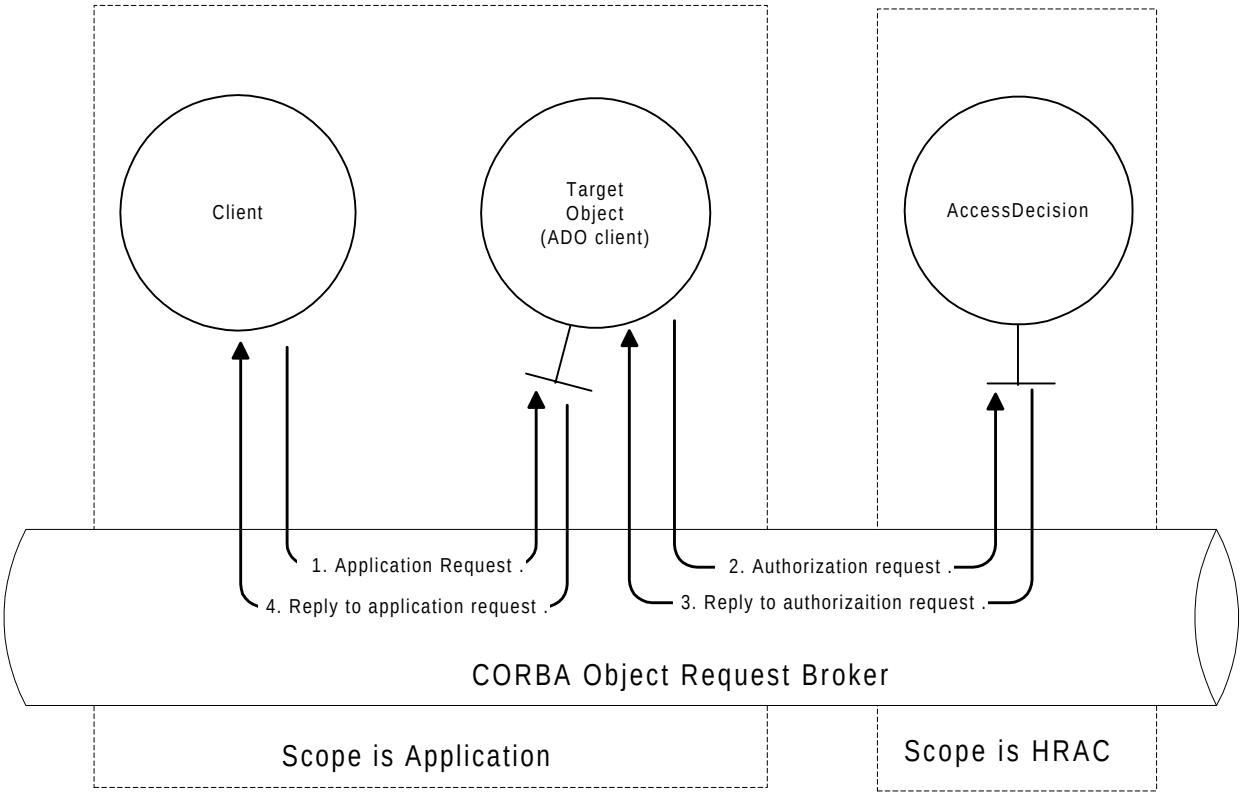


Figure 1-1 Interections among Client, Target and RAD

The sequence of the interaction, illustrated by Figure 1-1, is as follows:

1. An application client invokes an operation of the interface provided by the target object. The object request broker transfers this request to the target object and causes invocation of the appropriate method in the target object.
2. While processing the request, the target object requests authorization decision(s) from the Access Decision object by invoking the access_allowed() method of the ADO.
3. The Access Decision object consults other objects that are internal to the RAD (described in this submission) to make an access decision. The access decision is returned to the Target Object (ADO client) as a boolean.
4. The target object, after receiving an authorization decision, is responsible for enforcing the decision. If access was granted by the ADO, the target object performs the requested operation and returns the results. If access to secured resources was denied, the target object may return partial results or raise an exception to the Client.

A detailed description of the object model and design of the ADO (and its interaction with other RAD objects) can be found in Section 2.3 of this submission.

1.2.2 Reference Models

Two views of the RAD are presented in the following models. The first is the access decision model. This represents the relationship of objects involved in making an access decision. The second view is the Administrative view and represents how an RAD is configured. Administration of Access Policy is beyond the scope of the RAD and is clearly indicated as such on this model diagram.

The Resource Access Decision facility reference model defines a framework within which a wide variety of access control polices may be supported. The reference models below clearly indicate the scope of this submission response by heavy dotted lines. In some cases there are types that occur within the scope of this response that represents concepts and/or services that lie beyond the scope of the RAD. An example of this is the concept of a "secured resource" which is only represented within the scope of the RAD by a ResourceName. Where this occurs these external concepts appear in the model, but outside the dotted line to aid the reader in an understanding of the relationship between the RAD and the external concepts and/or services. The appearance of objects outside the scope of the submission is conceptual and is presented only to aid in understanding the types that occur within the RAD.

RAD types that represent or encapsulate external concepts and/or services

ResourceName — A "secured resource" is represented within the RAD by a ResourceName that is a structure containing an AuthorityId for the namespace and a sequence of name/value pairs..

Operation — Secured resources have one or more operations which may be performed on them (such as create, get, set, use..). These operations are represented within the RAD as strings.

PolicyName — "Policy" (the rules used for controlling access to secured resources and their operations) is beyond the scope of the RAD, but when referenced within the RAD, is identified by a PolicyName that is a string.

DynamicAttributeService — The DynamicAttributeService may consult an external AttributeEvaluator.

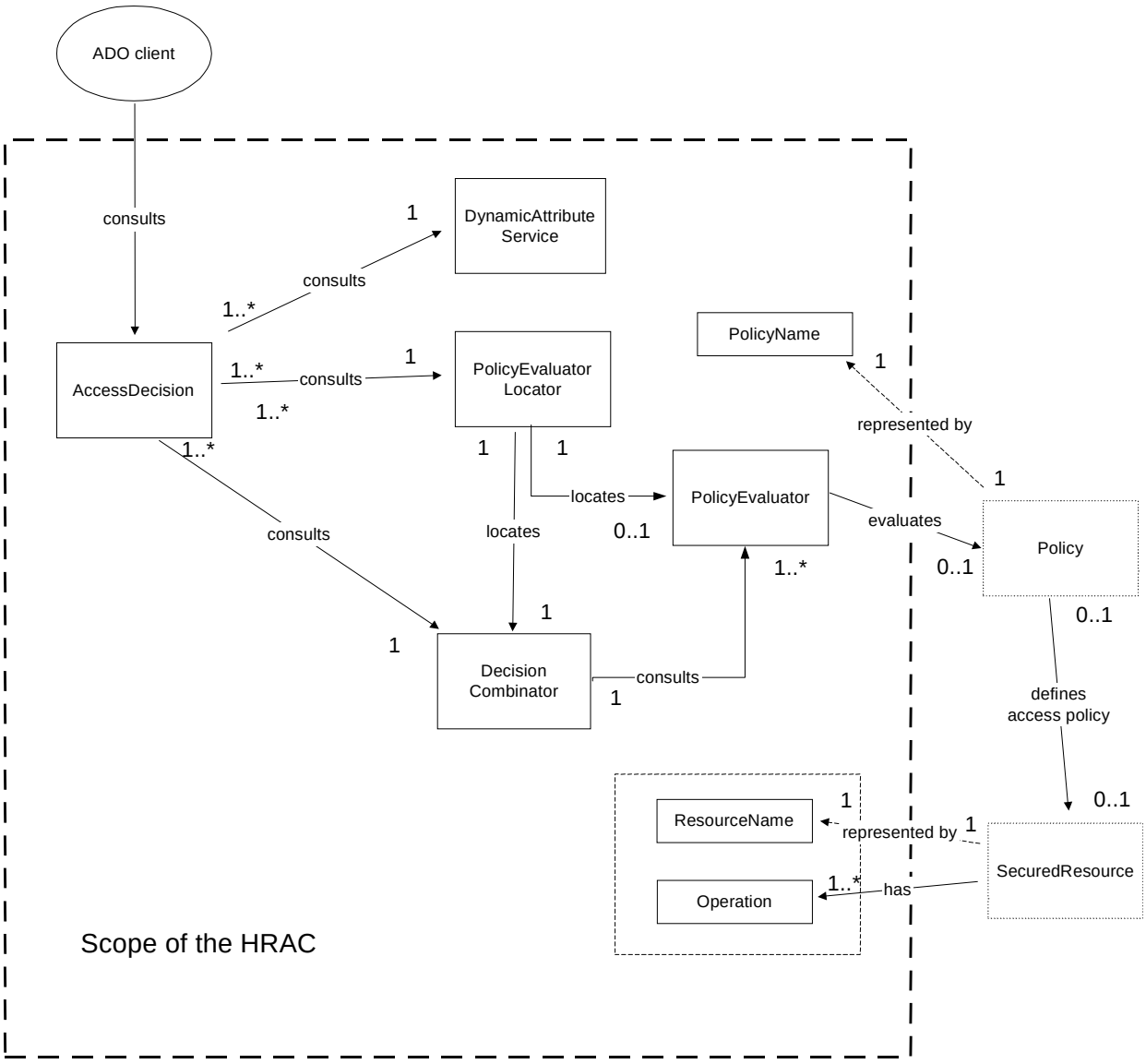


Figure 1-2 Access Decision Model

An Access Decision is requested by a client by invoking the access_allowed() method of the AccessDecision object (ADO) passing a ResourceName, Operation, and SecAttributes. The ADO consults a DynamicAttributeService to obtain an updated list of SecAttributes that include any dynamic attributes currently applicable for this access decision. The DynamicAttributeService may consult externally provided dynamic attribute evaluators as part of its implementation. The AccessDecision object also consults the PolicyEvaluatorLocator to obtain object references for the

PolicyEvaluator(s) and the DecisionCombinator that are required for an access decision. The AccessDecision object consults the DecisionCombinator that consults with any PolicyEvaluators responsible for interpreting access policy that controls access to the ResourceName/operation. The DecisionCombinator encapsulates policy combination logic and is responsible for understanding the policy that controls how a series of results from PolicyEvaluators are combined including any precedence rules that may apply. It is the response from the DecisionCombinator that is returned to the client.

1.2.3 Administrative Model

An Access Decision is requested by a client by invoking the access_allowed() method of the AccessDecision object (ADO) passing a ResourceName, Operation, and SecAttributes. The ADO consults a DynamicAttributeService to obtain an updated list of SecAttributes that include any dynamic attributes currently applicable for this access decision. The DynamicAttributeService may consult externally provided dynamic attribute evaluators as part of its implementation. The AccessDecision object also consults the PolicyEvaluatorLocator to obtain object references for the PolicyEvaluator(s) and the DecisionCombinator that are required for an access decision. The AccessDecision object consults the DecisionCombinator that consults with any PolicyEvaluators responsible for interpreting access policy that controls access to the ResourceName/operation. The DecisionCombinator encapsulates policy combination logic and is responsible for understanding the policy that controls how a series of results from PolicyEvaluators are combined including any precedence rules that may apply. It is the response from the DecisionCombinator that is returned to the client.

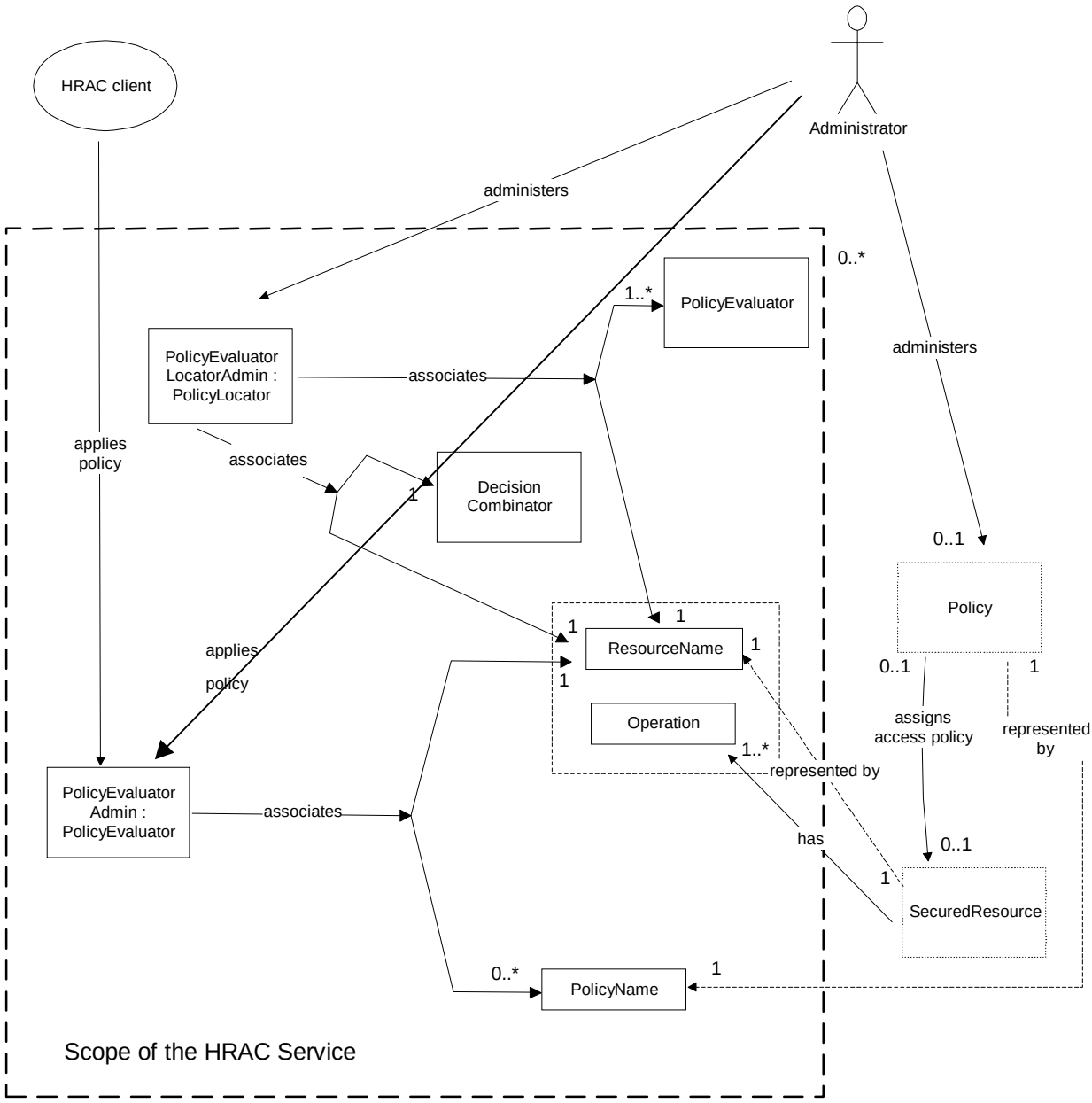


Figure 1-3 RAD Administrative Model

The administrative model of RAD is designed to allow replaceable RAD objects within an implementation and to allow RAD clients to apply previously defined policy to resources.

The administrative model is not intended to provide the Administrative interfaces necessary to define access policy. The definition of access policy (the rules that govern access to secured resources/operations) is outside the scope of this submission. This Administrative model clearly indicates this by placing Policy administration outside the dotted line that delineates the scope of the RAD submission.

The administrative interfaces of PolicyEvalautor are used to associate PolicyEvaluators and DecisionCombinators with a ResourceName. Multiple PolicyEvaluators may be associated with a single ResourceName. These evaluators will all be consulted during access decisions. There is only one DecisionCombinator provided for a ResourceName. This combinator is responsible for taking the results of the PolicyEvaluators evaluate() method and making a final access decision. PolicyEvaluators have an endless series of options for implementation. For this reason, the interface is public and evaluators may be “plugged-in” to an RAD framework by vendors and/or users. In the same sense, there are many possible policies for combining policy decisions. Some secured resources should not be accessible unless all the PolicyEvaluators return ACCESS_DECISION_ALLOWED. Other secured resources may be accessible if any one of the PolicyEvaluators allow access. Defining an interface for the DecisionCombinators allows custom combinators to be configured for a secured resource. It is possible to assign a default DecisionCombinator.

The PolicyEvaluatorBasicAdmin, PolicyEvaluatorNameAdmin and PolicyEvaluatorPatternAdmin interfaces are used to apply an existing named access policy to a secured resource. An application that wished to dynamically apply policy to newly created resources would be required to specify the names of those policies. The policy would be configured by an administrator using the administrative interfaces of the underlying access policy system and the required name associated with it (this is outside the scope of the RAD admin interfaces). Once this had been accomplished, a RAD client could apply this named policy using the PolicyName to a ResourceName. The PolicyEvaluatorBasicAdmin allows default policy to be assigned “by name” and a list of existing PolicyNames can be retrieved via the interface.

1.2.4 Information Model

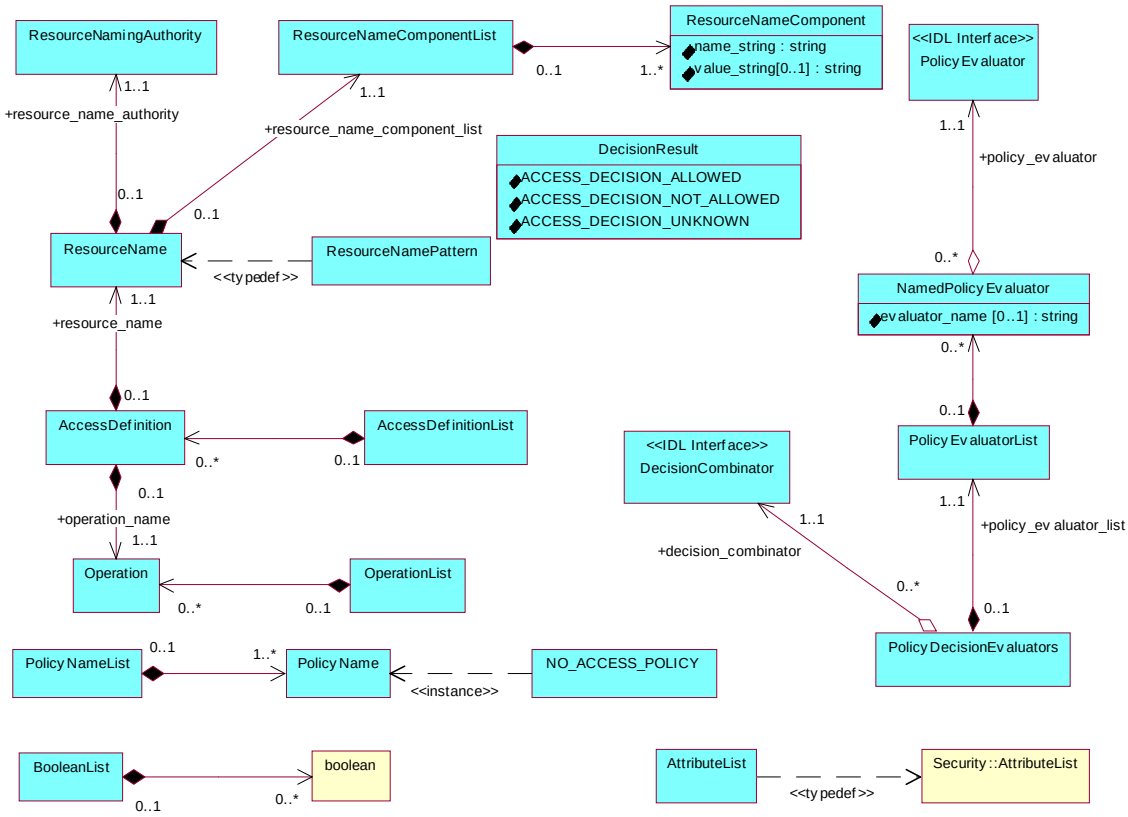


Figure 1-4 RAD Information Model

The information model of RAD is designed to be simple to implement and to use. The information model will be discussed in detail in Section Types.

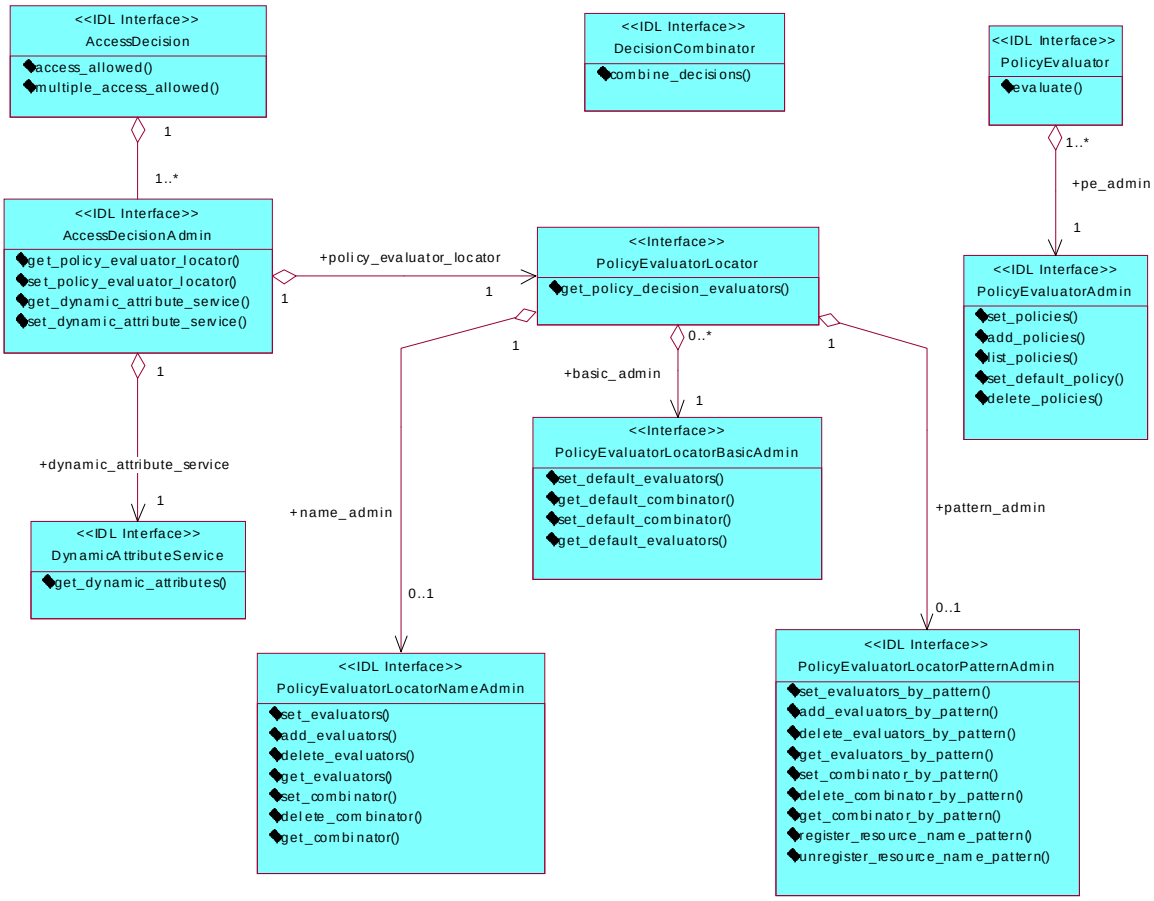


Figure 1-5 Computational Model

The computational model of RAD consists of two interface groups:

1. Runtime interfaces: AccessDecision, DynamicAttributeService, PolicyEvaluator, PolicyEvaluatorLocator, and DecisionCombinator.

2. Administrative interfaces: AccessDecisionAdmin, PolicyEvaluatorAdmin, and PolicyEvaluatorLocatorBasicAdmin, PolicyEvaluatorLocatorNameAdmin, PolicyEvaluatorLocatorPatternAdmin.

Among runtime interfaces, AccessDecision, PolicyEvaluatorLocator, and DynamicAttributeService are singletons, i.e. one instance of each interface is available in every implementation of RAD. On the other hand more than one instance of DecisionCombinator and PolicyEvaluator may be available.

The computational model will be discussed in detail in Section DfResourceAccessDecision module .

1.2.5 The Scope of the Specification

The specification addresses the following scope issues:

1. separation of access control logic from the application,
2. a standard interface for requesting access control decisions,
3. replaceable authorization engines (PolicyEvaluatorLocator, PolicyEvaluatorLocatorAdmin, and PolicyEvaluator),
4. custom integration of multiple authorization engines (PolicyEvaluatorLocatorAdmin and DecisionCombinator),
5. use of dynamic attributes in access decisions (DynamicAttributeService),
6. the application of pre-defined access policy to a resource. (PolicyEvaluatorAdmin).

1.3 DfResourceAccessDecision module

```
//File: DfResourceAccessDecision.idl
//

#ifndef _DF_RESOURCE_ACCESS_DECISION_IDL_
#define _DF_RESOURCE_ACCESS_DECISION_IDL_

#include "Security.idl"

#pragma prefix "omg.org"

module DfResourceAccessDecision {

interface AccessDecision {
…
};

interface DynamicAttributeService {
…
};

interface PolicyEvaluatorLocator {
…
};

interface DecisionCombinator {
…
};

interface PolicyEvaluator {
…
};

interface AccessDecisionAdmin {
…
};

interface PolicyEvaluatorLocatorAdmin {
…
};

interface PolicyEvaluatorAdmin {
…
};
```

The DfResourceAccessDecision contains four interfaces defined below and has type dependencies on the CORBA Security Service module.

#include <Security.idl>

The types declared within the Security service and used by the RAD are:

Security::AttributeList

These types are used for consistency with CORBASec and have the same meaning when used in RAD interfaces. They are typedef'd in this specification for ease of use.

#pragma prefix "omg.org"

In order to prevent name pollution and name clashing of IDL types this module (and all modules defined in this specification) uses the pragma prefix that is the omg DNS name.

1.3.1 Types

There are a number of structured types used widely through out the DfResourceAccessDecision Model. These types are described in this section:

Basic Types & Types used from the CORBA Security Service

```
//*************************************************************
//        Basic Types
//*************************************************************
```

typedef sequence<boolean> BooleanList;

typedef Security::AttributeList AttributeList;

BooleanList

A sequence of boolean used as a return value when multiple decisions are requested. This type is used as a return value in the multiple_access_allowed() method of the AccessDecision interface.

AttributeList

The Security::AttributeList is defined as follows in CORBA Security 1.2 (ptc/98-01-02). The AttributeList is provided as an input parameter by the "application" client when a request for an access decision is made. The AttributeList used for access decisions may be modified to include dynamic attributes by use of the get_dynamic_attributes() method of the DynamicAttributeService interface. As a convenience to the reader, the structure of a Security::AttributeList is replicated below.

typedef sequence<octet> Opaque;

```
// security attributes
   typedef unsigned long SecurityAttributeType;


struct ExtensibleFamily {
   unsigned short       family_definer;
   unsigned short       family;
};
struct AttributeType {
     ExtensibleFamily     attribute_family;
     SecurityAttributeType attribute_type;
};


struct SecAttribute {
   AttributeType        attribute_type;
   Opaque               defining_authority;
   Opaque               value;
   // the value of this attribute can be
   // interpreted only with knowledge of type
};

typedef sequence <SecAttribute> AttributeList;
```

Types that identify and manage information about secured resources

ResourceNameComponent

```
//*******************************************************
//  Types that identify a secured resource
//*******************************************************

struct ResourceNameComponent {
string   name_string;
string  value_string;
};
typedef sequence<ResourceNameComponent>
ResourceNameComponentList;

typedef string ResourceNamingAuthority;

struct ResourceName {
ResourceNamingAuthority resource_naming_authority;
ResourceNameComponentList resource_name_component_list;
};

typedef ResourceName  ResourceNamePattern;

typedef string      Operation;
typedef sequence<Operation> OperationList;
```

A datum element of this type is invalid if the name_string member has empty value.

ResourceNameComponentList

A datum element of type ResourceNameComponentList is invalid if it is empty or any of its sub-elements is invalid.

ResourceNamingAuthority

A ResourceNamingAuthority is used to identify an authority whose defined the semantics of the naming scheme used in the components of the corresponding resource_name_component_list data member.

ResourceName

A ResourceName is used to identify a **secured resource**. A ResourceName contains a unique identifier for the naming authority and a sequence of ResourceNameComponents. Each ResourceNameComponent includes a name and value string. This combination of naming authority and name/value pairs allows for categorization and grouping of resources if desired.

A datum of type ResourceName is invald if either resource_name_authority or resource_name_component_list is invalid.

ResourceNamePattern

A ResourceNamePattern is used in Administrative interfaces to allow generalized regular expressions to be provided in the value_string of a ResourceNameComponent for the purpose of administering groups of secured resources. The regular expression syntax is defined by 9945-2:1993 (ISO/IEC) Information Technology-Portable Operating System Interface (POSIX)-Part2: Shell and Utilities IEEE/ANSI Std 1003.2-1992 & IEEE/ANSI 1003.2a-1992 Section 2.8, pages 77-91, "Regular Expression Notation".

A datum of type ResourceNamePattern is invald if either resource_name_authority or resource_name_component_list is invalid.

Operation

A datum element of this type is invalid if it is empty.

OperationList

An OperationList is used to identify a list of operations that may be performed on a secured resource.

Types associated with evaluating Access Policy

```
//*****************************************************
//  Types associated with evaluating Access Policy
//*****************************************************
typedef string     PolicyName;
typedef sequence<PolicyName> PolicyNameList;

const PolicyName NO_ACCESS_POLICY = "NO_ACCESS_POLICY";

struct NamedPolicyEvaluator {
string    evaluator_name;
PolicyEvaluator  policy_evaluator;
};
typedef sequence<NamedPolicyEvaluator> PolicyEvaluatorList;

struct PolicyDecisionEvaluators {
PolicyEvaluatorList policy_evaluator_list;
DecisionCombinator    decision_combinator;
};
```

PolicyName

A PolicyName is a string used to identify an access policy for a secured resource. This type is only used in the PolicyEvalutorAdmin interface. It is used as an input parameter to the replace_policy(), add_policy(), and set_default_policy() methods of the PolicyEvaluatorAdmin interface. PolicyNames are assigned by the administrative

interface of the policy engine and cannot be modified or controlled by the RAD. There is one standard PolicyName of “NO_ACCESS_POLICY”. See the PolicyEvaluatorAdmin interface for usage.

A datum element of this type is invalid if it is empty.

PolicyNameList

A PolicyNameList is a sequence of PolicyNames. It is returned from the list_policy() method of the PolicyEvaluatorAdmin interface.

A datum element of this type is invalid if it is empty or any of its sub-elements is invalid.

NamedPolicyEvaluator

A NamedPolicyEvaluator is a structure that contains the name of the Policy Evaluator and the object reference for the policy evaluator. The evaluator_name will be null in implementations that choose not to name evaluators. Providing named evaluators allows an implementation to apply precedence logic based on evaluator names when making an access decision.

A datum element of type NamedPolicyEvaluator is invalid if its data member “policy_evaluator” has value *nil*.

PolicyEvaluatorList

A PolicyEvaluatorList is a sequence of NamedPolicyEvaluator. The administrative interfaces of PolicyEvaluatorLocator interface allow the association of a list of NamedPolicyEvaluator(s) with a ResourceName. This type is returned from get_policy_decision_evaluators() and set_default_evaluators() and is used as an input parameter in the set_evaluators(), add_evaluators(), delete_evaluators(),set_evaluators_by_pattern(), add_evaluators_by_pattern(), delete_evaluators_by_pattern(),and set_default_evaluators()operations. The PolicyEvaluatorList returned from the PolicyEvaluatorLocator is passed to the DecisionCombinator returned from the PolicyEvaluatorLocator.

A datum element of type PolicyEvaluatorList is invalid if it is empty or any of its elements is invalid.

PolicyDecisionEvaluators

The PolicyDecisionEvaluators struct contains a PolicyEvaluatorList and the DecisionCombinator. This is the type returned from the get_policy_decision_evaluators() operations of the PolicyEvaluatorLocator interface. This structure contains the references of all the objects that may be consulted during an access decision.

1.3.2 Types used to request access decisions

```
//*****************************************************
//      Types used to request an Access Decision
//*****************************************************

struct AccessDefinition {
ResourceName resource_name;
Operation  operation;
};
typedef sequence<AccessDefinition> AccessDefinitionList;

enum DecisionResult {ACCESS_DECISION_ALLOWED,
 ACCESS_DECISION_NOT_ALLOWED,
 ACCESS_DECISION_UNKNOWN
};
```

AccessDefinition

The AccessDefinition struct is provided to allow multiple access definitions to be defined. It contains the ResourceName and the operation name for the secured resource access being requested. AccessDefinition is used as an input parameter to the access_allowed() method of the AccessDecision interface and the evaluate() method of the PolicyEvaluator interface.

A datum element of this type is invalid if either of its members is invalid.

AccessDefinitionList

AccessDefinitionList is the type used to request multiple access decisions in a single operation. It is used as an input parameter to the multiple_access_allowed() method of the AccessDecision interface and the multiple_evaluate() method of the PolicyEvaluator interface.

DecisionResult

DecisionResult is an enum with three possible values. The values are:

ACCESS_DECISION_ALLOWED
: the policy evaluated for this ResourceName, operation and Attribute list indicates that access is ALLOWED.

ACCESS_DECISION_NOT_ALLOWED
: the policy evaluated for this ResourceName, operation and Attribute list indicates access is NOT_ALLOWED.

ACCESS_DECISION_UNKNOWN
: the policy evaluated for this ResourceName, operation and Attribute list indicates an access decision cannot be made.

This type is used as a result in access decisions where access policy is applied. This is the type returned from the evaluate() method of the PolicyEvaluator.

Exceptions

The following exceptions are used in this module

```
//*******************************************************
//*          Exception Data types
//*******************************************************
struct ExceptionData {
short  error_code;
string reason;
};
enum InternalErrorType {Fatal, NotFatal};

//*******************************************************
//    Exception thrown by the Access Decision Object
//*******************************************************

exception InternalError{InternalErrorType et;};

//*******************************************************
//    Exception thrown by Internal non-admin interfaces
//*******************************************************

exception ComponentError{
ExceptionData ed;
InternalErrorType et;
};

//*******************************************************
//        Exceptions thrown by Admin Interfaces
//*******************************************************

exception PatternConflict {ExceptionData ed;};
exception PatternDuplicate {ExceptionData ed;};
exception PatternNotRegistered {ExceptionData ed;};
exception PatternInUse {ExceptionData ed;};
exception InputFormatError {ExceptionData ed;};
exception ResourceNameNotFound {ExceptionData ed;};
exception NoAssociation {ExceptionData ed;};
exception InvalidPolicy {ExceptionData ed;};
exception DuplicateEvaluatorName {ExceptionData ed;};
exception InvalidResourceName {};
exception InvalidResourceNamePattern {};

exception InvalidPolicyEvaluatorList {
ExceptionData   ed;
NamedPolicyEvaluator first_invalid_element;
};

exception InvalidPolicyNameList {
ExceptionData ed;
PolicyName  first_invalid_element;
};
```

ExceptionData

The ExceptionData structure is included in most RAD exceptions. The contents of the error_code and reason are implementation dependent.

InternalError

The InternalError exception is reserved for internal logic errors and is not used as a reason code for rejecting a request. This is the only exception that is thrown by the AccessDecision object. Indicating Fatal means that the ADO client should discontinue using the ADO.

ComponentError

The ComponentError exception may be thrown by non-administrative interfaces to alert the AccessDecision object when a component encounters an internal error. If the ComponentError is Fatal, the AccessDecision object must determine if it can continue to process without the component. If it cannot, it must throw a InternalError with Fatal. If the Access Decision Object can continue to function without this component or if the exception error type was NotFatal, it is implementation dependent what the ADO returns to the client.

PatternConflict

The PatternConflict exception is thrown by the PolicyEvalutatorLocatorAdmin when a register_resource_name_pattern() detects a pattern that conflicts with an existing registered pattern and the implementation does not support conflicting patterns.

PatternDuplicate

The PatternDuplicate exception is thrown by the PolicyEvaluatorLocatorAdmin when an register_resource_name_pattern() detects a duplicate pattern registration.

PatternNotRegistered

The PatternNotRegistered exception is thrown by PolicyEvaluatorLocatorAdmin operations when an when an attempt is made to use a pattern in an administrative interface without registering the pattern first.

PatternInUse

The PatternInUse exception is thrown by PolicyEvaluatorLocatorAdmin unregister_resource_name_pattern when an attempt is made to unregister a pattern that is currently in use by the RAD.

InputFormatError

The InputFormatError exception is thrown by the administrative interface operations when an input parameter is provided in a format that is unacceptable to the RAD implementation. The error_code and reason are implementation dependent.

ResourceNameNotFound

The ResourceNameNotFound exception is thrown by PolicyEvaluatorAdmin interface operations when a ResourceName has not been defined. Not all implementations will require pre-definition of ResourceNames. For those implementations that do not require pre-definition, this exception will not be thrown.

NoAssociation

The NoAssociation exception is thrown by the PolicyEvaluatorAdmin interface delete_policies() operation when a an association between the ResourceName and PolicyName does not exist.

InvalidPolicy

The InvalidPolicy exception is thrown by the PolicyEvaluatorAdmin interface operations when an attempt is made to associate an Invalid PolicyName with a ResourceName or to set a default Policy that is invalid.

DuplicateEvaluatorName

The DuplicateEvaluatorName exception is thrown by the PolicyEvaluatorLocatorAdmin interface operations when an attempt is made to use those operations to add an evaluator that has the same value of its data member *evalautor_name* but different value of its data member *policy_evalautor* as some other named policy evalautor associated or to be associated (after the current operation was supposed to complete) with a resource name pattern.

InvalidResourceName

This exception is raised when the provided resource name is invalid. Please refer to the specification of type ResourceName for the description of valid and invalid datum elements of type ResourceName.

InvalidResourceNamePattern

This exceptions is thrown by corresponding operations when a resource name pattern, provided as an operation argument, has invalid syntax. Please refer to the specification of ResourceNamePattern data type for description of invalid values for ResourceNamePattern.

InvalidPolicyNameList

This exceptions is raised when the provided Policy NameList has invalid value. Please refer to the specification of PolicyNameList data type for a description of valid PolicyNameList datum elements.

first_invalid_element is first policy name in the PolicyNameList which caused the list to be invalid. If the value of this data member is nil than the list is invalid not because of a particular element, but because of some other reason (for example, because the list is empty).

InvalidPolicyEvaluatorList

This exception is raised when a policy evalautor list, passed as a part of an operation arguments, is invalid. Please refer to the specification of PolicyEvalautorList data type for a description of invalid PolicyEvalautorList datum elements of that type.

first_invalid_element is first named policy evalautor in the invalid list which caused the list to be invalid. If the value of this data member is nil than the list is invalid not because of a particular element, but because of some other reason (for example, because the list is empty).

1.3.3 AccessDecision interface

```
//*****************************************************
//    interface AccessDecision
//*****************************************************

interface AccessDecision {

    boolean access_allowed(
        in  ResourceName        resource_name,
        in Operation            operation,
        in  AttributeList       attribute_list
    )
        raises (InternalError);

    BooleanList multiple_access_allowed(
        in  AccessDefinitionList    access_requests,
        in  AttributeList           attribute_list
    )
        raises (InternalError);
};
```

The Access Decision object is used to request decisions on access based on a ResourceName, an Operation, and a list of SecAttributes. This submission provides a framework for the support of many policy evaluators. It is out of the scope of this submission to mandate how policy is defined or evaluated using the information provided by the client at the time access decisions are requested. This is the only interface that is necessary for a client to be familiar with in order to obtain access decisions from the RAD.

The AccessDecision object sometimes passes exceptions to callers indicating that it's encountered an internal error and is not able to make an access decision. This is different from the behavior of many operating systems, which have a default-deny or a default-grant policy when an internal failure occurs, but don't report the failure to their callers. This difference arises because RAD is an access decision service, not an access control service. In all cases, the application which calls RAD is responsible for enforcing the policy decision which RAD makes. Therefore, the RAD client application is the right place to make the policy enforcement decision about what should be done when RAD is not able to make a policy decision.

The AcceesDecisionObject is a singleton, i.e. only one instance of this interface is provided per each instance of RAD.

access_allowed()

A single access decision is requested and a boolean is returned. The InternalError exception is reserved for internal logic errors and should **not** be used as a reason code for rejecting a request. As a security consideration, ADO clients are not provided with the specific reason for not allowing access.

Preconditions

1. "resource_name" is valid.
2. "operation" is valid.

Postconditions

1. return == authorization decision for the requested operation on the specified resource name by a principal with the specified security attributes.

multiple_access_allowed()

Multiple access decisions are requested in a single method invocation and a sequence of booleans are returned. The boolean sequence maps one to one in the same order to the provided sequence of ResourceName/operation pairs. The InternalError exception is reserved for internal logic errors and should **not** be used as a reason code for rejecting a request. ADO clients are not exposed to the security reason for not allowing access. Indicating Fatal means that the ADO client should discontinue using the ADO

Preconditions

1. All elements of "access_request" are valid.
2. Each element of the returned list is an authorization decision for the corresponding request in the "access_requests" list. I.e. first element of the returned list is an authorization decision for the first element of access_request", and so on. Where an authorization decision == authorization decision for the requested operation on the specified resource name by a principal with the specified security attributet.
3. Each element of the returned list is an authorization decision for the corresponding request in the "access_requests" list. I.e. first element of the returned list is an authorization decision for the first element of access_request", and so on. Where an authorization decision == authorization decision for the requested operation on the specified resource name by a principal with the specified security attributes.

1.3.4 DynamicAttributeService interface

```
//*******************************************************
//    interface DynamicAttributeService
//*******************************************************

interface DynamicAttributeService {

    AttributeList get_dynamic_attributes(
        in AttributeList   attribute_list,
        in ResourceName resource_name,
        in  Operation  operation
    )
        raises (ComponentError);
};
```

4.

The DynamicAttributeService interface is used to obtain a new list of SecAttributes that are applicable to an access decision. This service may encapsulate calls to a relationship service and/or application specific logic to determine how the original AttributeList provided by the client should be modified.

get_dynamic_attributes()

This method takes the parameters provided by the client of the AccessDecision object; the AttributeList, the ResourceName, and the operation and determines what (if any) dynamic attributes should be added to the AttributeList. In addition, the returned AttributeList may be modified by this service. The service may add or remove SecAttributes to this list. It is the returned list of SecAttributes that is used as the basis of access decisions by the RAD.

Preconditions

1. "resource_name" is valid.

2. "operation" is valid.

Postconditions

No postconditions.

1.3.5 PolicyEvaluatorLocator interface

```
//*******************************************************
//    interface PolicyEvaluatorLocator
//*******************************************************

interface PolicyEvaluatorLocator {

    readonly attribute PolicyEvaluatorLocatorBasicAdmin basic_admin;

    readonly attribute PolicyEvaluatorLocatorNameAdmin name_admin;

    readonly attribute PolicyEvaluatorLocatorPatternAdmin pattern_admin;

    PolicyDecisionEvaluators get_policy_decision_evaluators(
        in ResourceName    resource_name
    )
    raises (ComponentError);
};
```

The PolicyEvaluatorLocator interface is used to locate the PolicyEvaluators and the DecisionCombinator associated with a ResourceName. This submission provides a framework for the support of one or more policy evaluators for a single resource.

readonly attribute PolicyEvaluatorLocatorBasicAdmin basic_admin

The PolicyEvaluatorLocator's basic administrative interface can be obtained via this attribute.

readonly attribute PolicyEvaluatorLocatorNameAdmin name_admin

The interface for administrating associations between resource names and policy evaluators as well as between resource names and decision combinators can be obtained via this attribute.

readonly attribute PolicyEvaluatorLocatorPatternAdmin pattern_admin

The interface for administrating associations between resource name patterns and policy evaluators as well as between resource name patterns and decision combinators can be obtained via this attribute. If an implementation of a policy evaluator locator does not implement support for resource name patterns this attribute must be *null*.

get_policy_decision_evaluators()

A PolicyDecisionEvaluators structure which contains a list of NamedPolicyEvaluator consisting of the names of the PolicyEvaluators and their object references, and the DecisionCombinator object reference for the resource is returned to the client.

Preconditions

1. "resource_name" is valid.

Postconditions

1. The returned references are not nil.
2. No elements of “policy_evaluator_list” in the returned datum have same value of “evaluator_name.”

1.3.6 DecisionCombinator interface

The DecisionCombinator interface is used to encapsulate the policy for the way that decisions of multiple PolicyEvaluators is combined. DecisionCombinators may be simple or arbitrarily complex. A default combinator may be used for all access decisions, or combinators may be chosen specifically for access decisions on specific secured resources.

```
//*********************************************************
//    interface DecisionCombinator
//*********************************************************

interface DecisionCombinator{

    boolean combine_decisions(
        in  ResourceName resource_name,
        in  Operation  operation,
        in  AttributeList attribute_list,
    in  PolicyEvaluatorList policy_evaluator_list
    )
    raises (ComponentError);
};
```

Functions consisting of a global combinator operator are easy to implement; an example of such a policy is:

AND ((Evaluator_1 = ACCESS_DECISION_ALLOWED),

(Evaluator_2 = ACCESS_DECISION_ALLOWED), ...)

This policy can be expressed as an application of a global combinator ("AND" in this case) to the results returned by ALL the PolicyEvaluator objects passed to the DecisionCombinator.

The thing which makes this kind of policy easy to implement is that it's not necessary to know anything about the result returned by any specific PolicyEvaluator object, and hence the PolicyEvaluator objects can all be treated the same and can be called in any order.

The disadvantages of this kind of policy are:

- They aren't very expressive (there are lots of kinds of real-world policies which can't be expressed using only a global combinator)

- They are inefficient. It's always necessary to call all the PolicyEvaluator objects passed to the DecisionCombinator object in order to make a decision. An important goal of the DecisionCombinator design is to support complex policies which can be efficiently evaluated. A policy like the following can't be expressed using only a global combinator, but should be implementable as a DecisionCombinator object:

  (Evaluator_1 result is ACCESS_DECISION_ALLOWED) OR

  ((Evaluator_2 result is ACCESS_DECISION_ALLOWED) AND

  (Evaluator_3 result is (ACCESS_DECISION_ALLOWED OR

  ACCESS_DECISION_UNKNOWN)))

Note that this policy can be short-circuit evaluated: if the DecisionCombinator calls Evaluator_1 and it returns ACCESS_DECISION_ALLOWED as a decision result, then it doesn't need to call Evaluator_2 and Evaluator_3 at all. However, In order to support evaluation of this policy, the DecisionCombinator object needs to be able to match the PolicyEvaluator objects passed to it as input to the formal parameters in this expression. This is why the DecisionCombinator interface accepts as input a structure containing both a reference to a PolicyEvaluator object and the name of that PolicyEvaluator object; it uses the PolicyEvaluator name to figure out which evaluators to call in which order; it uses the PolicyEvaluator object's reference to call the object and request a decision result, and then it uses the PolicyEvaulator object's name again to plug the decision result into the policy combinator expression above.

combine_decisions()

The DecisionCombinator is responsible for determining what PolicyEvaluators (from the list passed to it) must be called and how the results are to provide a boolean result. This is the result that will be returned by the AccessDecision object to the original client of the RAD facility.

Preconditions

1. "resource_name" is valid.
2. "operation" is valid.
3. "policy_evaluator_list" is valid.

Postconditions

1. No postconditions.

1.3.7 PolicyEvaluator interface

The PolicyEvaluator interface is used to obtain an access decision based on an encapsulated policy for the ResourceName/operation when provided a list of effective Security Attributes for the requestor. This submission provides a framework for the support of one or more policy evaluators for a single resource.

```
//*******************************************************
//    interface PolicyEvaluator
//*******************************************************

interface PolicyEvaluator {

readonly attribute PolicyEvaluatorAdmin pe_admin;

    DecisionResult evaluate(
        in  ResourceName resource_name,
        in  Operation  operation,
        in  AttributeList attribute_list
    )

    raises (ComponentError);

};
```

readonly attribute PolicyEvaluatorAdmin

If the PolicyEvaluator has an associated administrative interface, it can be obtained via this attribute. If an administrative interface is not available for this evaluator, this attribute will be nil.

evaluate()

A single access decision is requested based on access policy(s) this evaluator determines is appropriate for the named resource. The decision is based on the ResourceName, the operation, and the effective Security Attributes. The SecAttributes passed to the AccessDecision object by the client in access_allowed() may have been modified by the DynamicAttributeService get_dynamic_attributes() method before the PolicyEvaluator is called. The DecisionResult is a ternary result. The DecisionResult is as follows:

ACCESS_DECISION_ALLOWED
the policy evaluated for this ResourceName, operation and Attribute list indicates that access is ALLOWED.

ACCESS_DECISION_NOT_ALLOWED
the policy evaluated for this ResourceName, operation and Attribute list indicates access is NOT_ALLOWED.

ACCESS_DECISION_UNKNOWN
the policy evaluated for this ResourceName, operation and Attribute list indicates an access decision cannot be made.

Preconditions

1. "resource_name" is valid.

2. "operation" is valid.

Postconditions

1. return == authorization decision for the requested operation on the specified resource name by a principal with the specified security attributes.

1.3.8 AccessDecisionAdmin interface

The Access Decision Admin object is provided to allow a standard mechanism for replacement of the vendor provided PolicyEvaluatorLocator and the DynamicAttributeService.

```
//*****************************************************
//    interface AccessDecisionAdmin
//*****************************************************

interface AccessDecisionAdmin {

    PolicyEvaluatorLocator get_policy_evaluator_locator();

    void set_policy_evaluator_locator (

        in PolicyEvaluatorLocator policy_evaluator_locator

    );

    DynamicAttributeService get_dynamic_attribute_service();

    void set_dynamic_attribute_service(

        in DynamicAttributeService dynamic_attribute_service

    );
};
```

get_policy_evaluator_locator()

This operation returns the PolicyEvaluatorLocator used by the access decision object.

set_policy_evaluator_locator()

This operation sets the PolicyEvaluatorLocator used by the access decision object.

get_dynamic_attribute_service()

This operation returns the DynamicAttributeService used by the access decision object.

set_dynamic_attribute_service()

This operation sets the DynamicAttributeService used by the access decision object.

1.3.9 PolicyEvaluatorLocatorBasicAdmin interface

The PolicyEvaluatorLocatorBasicAdmin object is used to administrate default associations between PolicyEvaluators and ResourceNames as well as default associations between DecisionCombinators and ResourceNames.

```
//*******************************************************
//    interface PolicyEvaluatorLocatorBasicAdmin
//*******************************************************

interface PolicyEvaluatorLocatorBasicAdmin {

    PolicyEvaluatorList set_default_evaluators(
        in PolicyEvaluatorList policy_evaluator_list

    )
    raises (DuplicateEvaluatorName, InvalidPolicyEvaluatorList);

    PolicyEvaluatorList get_default_evaluators();

    DecisionCombinator get_default_combinator ();

    void set_default_combinator(
        in DecisionCombinator decision_combinator

    );

};
```

set_default_evaluators()

The list of PolicyEvaluators provided is set as the default evaluators for any ResourceName for which PolicyEvaluators have not been explicitly assigned. Default evaluators are overridden by the add_evaluators() or replace_evaluators() methods. The default evaluators will be returned by the PolicyEvaluatorLocator get_policy_decision_evaluators() method when no PolicyEvaluators have been explicitly assigned for a ResourceName.

Preconditions

No preconditions.

Postconditions

1. default_evaluators == “policy_evaluator_list”

get_default_evaluators()

The default set of policy evaluators provided is returned.

Preconditions

No preconditions.

Postconditions

1. return == default_evaluators.

get_default_combinator()

The DecisionCombinator provided is returned.

Preconditions

No preconditions.

Postconditions

1. return == default_combinator.

set_default_combinator()

The DecisionCombinator provided is set as a default. This combinator is now the combinator used when a DecisionCombinator has not been explicitly specified for a secured resource. This combinator will be returned by the PolicyEvaluatorLocator get_policy_decision_evaluators() method for these resources.

Preconditions

No preconditions.

Postconditions

1. default_combinator == “decision_combinator”.

1.3.10 PolicyEvaluatorLocatorNameAdmin interface

The PolicyEvaluatorLocatorNameAdmin object is used to associate PolicyEvaluators with a ResourceName. The object is also used to associate the appropriate DecisionCombinator with a ResourceName. This submission provides a framework for the support of one or more policy evaluators for a single resource.

```
//*********************************************************
//    interface PolicyEvaluatorLocatorNameAdmin
//*********************************************************

interface PolicyEvaluatorLocatorNameAdmin {

    PolicyEvaluatorList get_evaluators(
        in ResourceName resource_name
    )
    raises (InvalidResourceName);

    void set_evaluators (
        in PolicyEvaluatorList policy_evaluator_list,
        in   ResourceName resource_name
    )
    raises (
        InvalidPolicyEvaluatorList,
        InvalidResourceName,
        DuplicateEvaluatorName
    );
```

```
    void add_evaluators (
        in PolicyEvaluatorList policy_evaluator_list,
        in ResourceName resource_name
    )
    raises (
        InvalidResourceName,
        InvalidPolicyEvaluatorList,
        DuplicateEvaluatorName
    );

    void delete_evaluators (
        in PolicyEvaluatorList policy_evaluator_list,
        in  ResourceNamePattern resource_name
    )
    raises (
        InvalidResourceName,
        InvalidPolicyEvaluatorList,
        DuplicateEvaluatorName
    );

    DecisionCombinator get_combinator (
        in  ResourceNamePattern resource_name
    )
    raises (InvalidResourceName);

    void set_combinator (
        in  DecisionCombinator decision_combinator,
        in  ResourceNamePattern resource_name
    )
    raises (InvalidResourceName);

    void delete_combinator (
        in  ResourceNamePattern resource_name
    )
    raises (InvalidResourceName);
};
```

get_evaluators()

The list of PolicyEvaluators associated with the ResourceNamePattern is returned.

Preconditions

No preconditions.

Postconditions

1. return == "resource_name".registered_ evalautor_list

set_evaluators()

A list of PolicyEvaluators is assigned for the named resource. If the resource had existing PolicyEvaluators assigned, they are removed and the entire list is replaced with the ones provided in this method. The replacement of evaluators for a resource

which previously had none results in the added list of evaluators being the only evaluators consulted on an access decision (system default evaluators are no longer consulted unless a system default evaluator is a member of the replacement list).

These evaluators will be the PolicyEvaluators returned by the PolicyEvaluatorLocator get_policy_decision_evaluators() method.

Preconditions

No preconditions.

Postconditions

1. “resource_name”.registered_evaluator_list == policy_evaluator_list

add_evaluators()

A list of PolicyEvaluators is added to the list of evaluators for the named resource. These evaluators will be in the list of PolicyEvaluators returned by the PolicyEvaluatorLocator get_policy_decision_evaluators() method. The addition of evaluators to a ResourceName which previously had none results in the added list of evaluators being the only evaluators consulted on an access decision (system default evaluators are no longer consulted unless a system default evaluator is a member of the added list).

Preconditions

No preconditions.

Postconditions

1. “resource_name”.registered_evaluator_list == union (policy_evaluator_list, “resource_name”.registered_evaluator_list)

delete_evaluators()

The list of PolicyEvaluators is removed from the list of evaluators for the named resource. These evaluators will not be in the list of PolicyEvaluators returned by the PolicyEvaluatorLocator get_policy_decision_evaluators() method.

Preconditions

No preconditions.

Postconditions

1. for the "resource_name" : “resource_name”.registered_evaluator_list = “resource_name”.registered_evaluators - "policy_evaluator_list"

get_combinator()

The DecisionCombinator specified for the named resource is returned. If a combinator has not been specified for the ResourceName provided, the return will be nil (it will not return the default combinator).

Preconditions

No preconditions.

Postconditions

1. return == "resource_name".registered_ decision_combinator

set_combinator()

A DecisionCombinator is specified for the named resource. This combinator will be returned by the PolicyEvaluatorLocator get_policy_decision_evaluators() method. The DecisionCombinator provided replaces any previous combinator specified for the secured resource.

Preconditions

No preconditions.

Postconditions

1. "resource_name".registered_ decision_combinator == “decision_combinator”

delete_combinator()

The DecisionCombinator for the ResourceName is removed. The default combinator will now be returned by the PolicyEvaluatorLocator get_policy_decision_evaluators() method.

Preconditions

No preconditions.

Postconditions

1. Resource names matching only "resource_name" will be associated with the default combinator.

1.3.11 PolicyEvaluatorLocatorPatternAdmin interface

The PolicyEvaluatorLocatorPatternAdmin object is used to associate PolicyEvaluators with a ResourceNamePatterns. The object is also used to associate the appropriate DecisionCombinator with the ResourceNamePattern. This submission provides a framework for the support of one or more policy evaluators for a single resource pattern.

Patterns are used to group resource names without requiring the PolicyEvaluatorLocator administrator to enumerate all the resources names individually; this is accomplished by associating lists of PolicyEvaluator objects with ResourceNamePatterns, and checking whether a supplied resource name matches any of the Patterns with which it has associated PolicyEvaluators. This section describes how RAD objects decide whether a Pattern matches a resource name. Throughout the section, we use the shorthand phrase "exactly matches" to mean "is exactly the same string as". Patterns have a specific format:

- A Pattern must include a ResourceNamingAuthority.
- A Pattern must include a list of ResourceNameComponent strings.
- Each ResourceNameComponent consists of a name_string and a value_string.

Two kinds of ResourceNameComponents can occur in a pattern. The first kind is a component value pattern. It has the form:

- name_string is a string
- value_string is a regular expression

A resource name component matches a component value pattern only if its

- name_string exactly matches the pattern's name_string and its value_string matches the component value pattern's value_string regular expression.

The second kind of ResourceNameComponent which can occur in a pattern is a component wildcard pattern:

- name_string is "*"
- value_string is "*"

Every component of a resource name matches a component wildcard pattern.

A resource name matches a pattern if and only if the algorithm shown in the figure below returns MATCH.

The algorithm has two inputs: resource name (“name”) and resource name pattern (“pattern”). It also assumes availability of two functions:

SIZE – returns number of elements in a sequence,

MATCHES_AS_GRE – returns “yes” if the string provided by first argument matches another string provided by second argument, where the second string is interpreted according to regular expression syntax specified in the definition of ResourceNamePattern type on Page Types that identify and manage information about secured resources, otherwise “no”.

```
//*********************************************************
//    interface PolicyEvaluatorLocatorPatternAdmin
//*********************************************************

interface PolicyEvaluatorLocatorPatternAdmin {

    void register_resource_name_pattern(
        in  ResourceNamePattern pattern
    )
    raises (
        InvalidResourceNamePattern,
        PatternDuplicate,
        PatternConflict
    );

    void unregister_resource_name_pattern(
        in  ResourceNamePattern pattern
    )
    raises (
        InvalidResourceNamePattern,
        PatternNotRegistered,
        PatternInUse
    );
```

```
PolicyEvaluatorList get_evaluators_by_pattern (
    in ResourceNamePattern pattern
)
raises (
    InvalidResourceNamePattern,
    PatternNotRegistered
);

void set_evaluators_by_pattern (
    in PolicyEvaluatorList policy_evaluator_list,
    in   ResourceNamePattern pattern
)
raises (
    InputFormatError,
    PatternNotRegistered,
    DuplicateEvaluatorName
);

void add_evaluators_by_pattern (
    in PolicyEvaluatorList policy_evaluator_list,
    in ResourceNamePattern pattern
)
raises (
    InvalidResourceNamePattern,
    PatternNotRegistered,
    InvalidPolicyEvaluatorList,
    DuplicateEvaluatorName
);

void delete_evaluators_by_pattern (
    in PolicyEvaluatorList policy_evaluator_list,
    in  ResourceNamePattern pattern
)
raises (
    InvalidResourceNamePattern,
    PatternNotRegistered,
    InvalidPolicyEvaluatorList,
    DuplicateEvaluatorName
);

DecisionCombinator get_combinator_by_pattern (
    in  ResourceNamePattern pattern
)
raises (
    InvalidResourceNamePattern,
    PatternNotRegistered
);
```

```
    void set_combinator_by_pattern (
        in  DecisionCombinator decision_combinator,
        in  ResourceNamePattern pattern
    )
    raises (
        InvalidResourceNamePattern,
        PatternNotRegistered
    );

    void delete_combinator_by_pattern (
        in  ResourceNamePattern pattern
    )
    raises (
        InvalidResourceNamePattern,
        PatternNotRegistered
    );
};
```

register_resource_name_pattern()

Before a ResourceNamePattern can be used in the administrative interfaces, it must be registered. This allows the administration of name patterns separately from the administration of the association of patterns to evaluators and combinators. Since a ResourceName is a ResourceNamePattern, ResourceNames must also be registered.

Implementations may or may not support overlapping patterns; that is, an implementation may choose to allow registration of two patterns both of which match at least one name, or they may choose not to allow such registrations. An implementation which does not support overlapping patterns shall raise the PatternConflict exception when this method is used to register a pattern which overlaps with another previously registered pattern. Implementors should document whether their implementations support overlapping patterns or not.

Preconditions

No preconditions.

Postconditions

1. "resource_name_pattern" is registered.

unregister_resource_name_pattern()

ResourceNamePatterns may be unregistered. A ResourceNamePattern must not have any evaluators or combinators associated with it when it is unregistered.

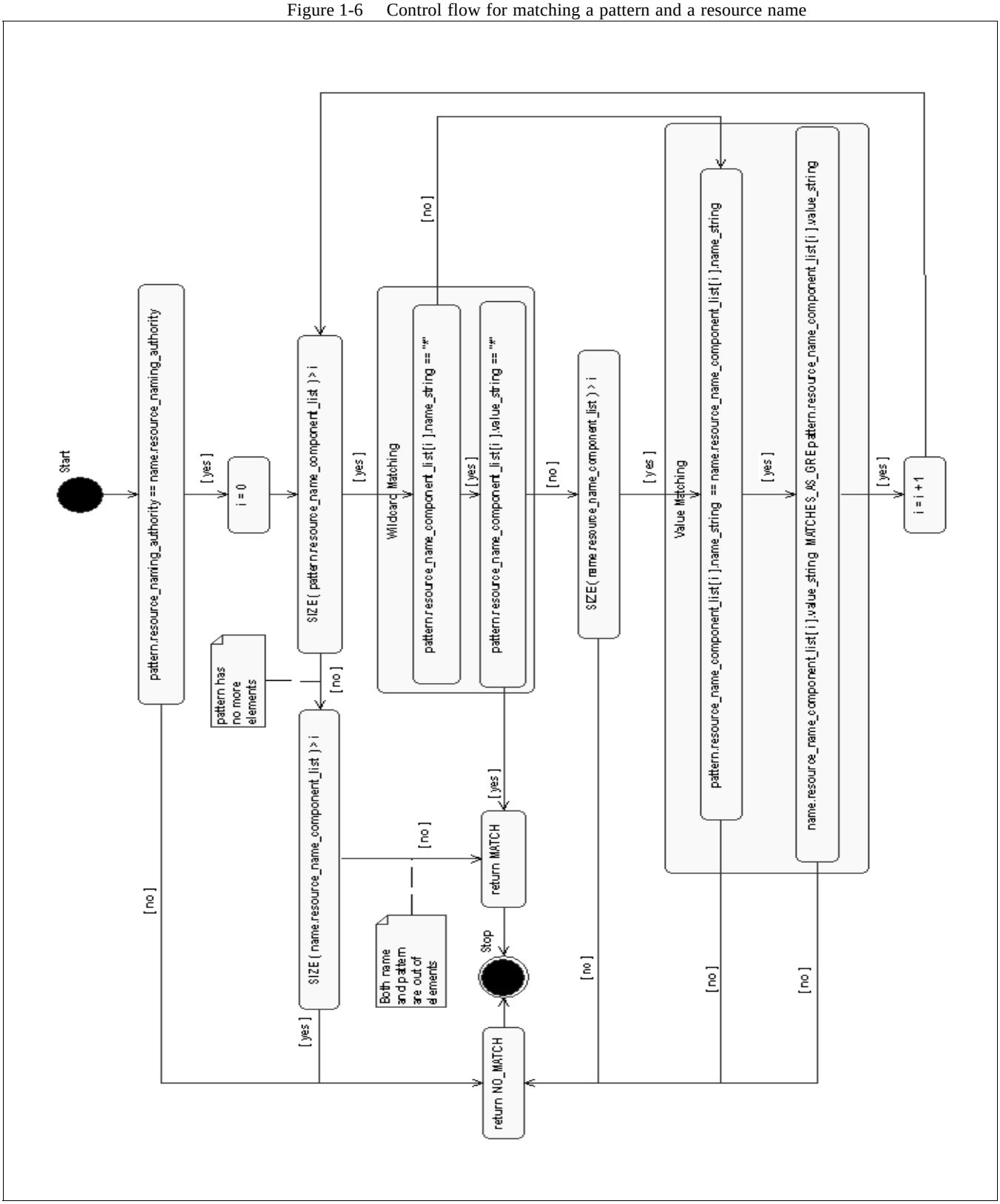
Preconditions

No preconditions.

Postconditions

1. "resource_name_pattern" is unregistered.

Figure 1-6 Control flow for matching a pattern and a resource name



get_evaluators_by_pattern ()

The list of PolicyEvaluators associated with the ResourceNamePattern is returned.

Preconditions

No preconditions.

Postconditions

1. return == "resource_name_pattern".registered_ evalautor_list

set_evaluators_by_pattern ()

A list of PolicyEvaluators is assigned for the resources that will match ResourceNamePattern. If the resource had existing PolicyEvaluators assigned, they are removed and the entire list is replaced with the ones provided in this method. The replacement of evaluators for a resource which previously had none results in the added list of evaluators being the only evaluators consulted on an access decision (system default evaluators are no longer consulted unless a system default evaluator is a member of the replacement list).

These evaluators will be the PolicyEvaluators returned by the PolicyEvaluatorLocator get_policy_decision_evaluators() method.

Preconditions

No preconditions.

Postconditions

1. “resource_name_pattern”.registered_evaluator_list == policy_evaluator_list

add_evaluators_by_pattern ()

A list of PolicyEvaluators is added to the list of evaluators for the resources that will match ResourceNamePattern. These evaluators will be in the list of PolicyEvaluators returned by the PolicyEvaluatorLocator get_policy_decision_evaluators() method. The addition of evaluators to a ResourceNamePattern which previously had none results in the added list of evaluators being the only evaluators consulted on an access decision (system default evaluators are no longer consulted unless a system default evaluator is a member of the added list).

Preconditions

No preconditions.

Postconditions

1. “resource_name_pattern”.registered_evaluator_list == union (policy_evaluator_list, “resource_name_pattern”.registered_evaluator_list)

delete_evaluators_by_attern ()

The list of PolicyEvaluators is removed from the list of evaluators for the resources that will match ResourceNamePattern. These evaluators will not be in the list of PolicyEvaluators returned by the PolicyEvaluatorLocator get_policy_decision_evaluators() method.

Preconditions

No preconditions.

Postconditions

1. for the "resource_name_pattern" : “resource_name_ pattern”.registered_evaluator_list = “resource_name_ pattern”.registered_evaluators - "policy_evaluator_list"

get_combinator_by_pattern ()

The DecisionCombinator specified for by the ResourceNamePattern is returned. If a combinator has not been specified for the ResourceNamePattern provided, the return will be nil (it will not return the default combinator).

Preconditions

No preconditions.

Postconditions

1. return == "resource_name_pattern".registered_ decision_combinator

set_combinator_by_pattern ()

A DecisionCombinator is specified for the resources that will match ResourceNamePattern. This combinator will be returned by the PolicyEvaluatorLocator get_policy_decision_evaluators() method. The DecisionCombinator provided replaces any previous combinator specified for the secured resource.

Preconditions

No preconditions.

Postconditions

1. "resource_name_pattern".registered_ decision_combinator == “decision_combinator”

delete_combinator_by_pattern ()

The DecisionCombinator for the ResourceNamePattern is removed. The default combinator will now be returned by the PolicyEvaluatorLocator get_policy_decision_evaluators() method for those resources that used to match the specified ResourceNamePattern and do not match any other ResourceNamePattern set by set_combinator_by_pattern() operation.

Preconditions

No preconditions.

Postconditions

1. Resource names matching **only** "resource_name_pattern" will be associated with the efault combinator.

1.3.12 PolicyEvaluatorAdmin interface

```
//*******************************************************
//    interface PolicyEvaluatorAdmin
//*******************************************************

interface PolicyEvaluatorAdmin {

    void set_policies(
        in PolicyNameList policy_names,
        in ResourceName  resource_name
    )
    raises (
        InvalidResourceName,
        ResourceNameNotFound,
        InvalidPolicyNameList
    );

    void add_policies(
        in PolicyNameList policy_names,
        in ResourceName resource_name
        )
    raises (
        InvalidResourceName,
        ResourceNameNotFound,
        InvalidPolicyNameList
    );

    void delete_policies(
        in PolicyNameList policy_names,
        in ResourceName resource_name
    )
    raises (
        InvalidResourceName,
        ResourceNameNotFound,
        InvalidPolicyNameList,
        NoAssociation
    );

    PolicyNameList list_policies();

    PolicyName set_default_policy(
        in PolicyName policy_name
)
    raises (
        InvalidPolicy
    );

};
```

The PolicyEvaluatorAdmin interface is used to associate named access policies with secured resources. It is assumed that the administrative tool used to create and manage access policies (outside the scope of this submission) provides a mechanism to allow policies to be associated with “names” which are represented as PolicyName (a string). This PolicyEvaluatorAdmin interface allows those policies to be applied “by name” to a secured resource represented by a ResourceName.

This interface is primarily provided for the application that wishes to assign a policy to a newly created resource programatically at the time of resource creation. It does, however, require that the application have knowledge of the named policies in order to choose an appropriate policy for access decisions.

set_policies()

The policies identified by PolicyNameList is associated with the secured resource identified by the ResourceName. If a single PolicyName of NO_ACCESS_POLICY is specified, then all policy is removed for the resource. If a PolicyNameList is applied to a ResourceName that has existing policy, then the policy will be replaced by the policy identified by this PolicyNameList.

Preconditions

No preconditions.

Postconditions

1. "resource_name".applied_policie_names == "policy_names".

add_policies()

The policy identified by PolicyNameList is added to the list of policies used when making access decisions for the secured resource identified by the ResourceName. If a PolicyNameList is added to a resource that has existing policy, then the policy will be added to the list of policies that control access decisions for the resource. An implementation is not required to support multiple policies for a resource. If the implementation does not support the application of multiple policies, then a InvalidPolicy exception shall be thrown for this method.

Preconditions

No preconditions.

Postconditions

1. "resource_name".applied_policy_names == union ("resource_name".applied_policy_names, "policy_names")

list_policies()

A list of names of all policies supported by this instance of PolicyEvaluator is returned to the client.

Preconditions

No preconditions.

Postconditions

1. return == all_existing_policy_names.

set_default_policy()

The policy identified by PolicyName is associated (as default) with any secured resource which has not yet been assigned an access policy.

Preconditions

No preconditions.

Postconditions

1. return == default_policy_name

1. return == default_policy_name

2. default_policy_name == "policy_name"

3. default_policy_name == "policy_name"

1.4 Conformance Classes

There are two conformance classes: “RAD without Patterns” and “RAD with Patterns”.

An implementation of Resource Access Decision (RAD) facility compliant to conformance class “**RAD without Patterns**” must implement all of the interfaces defined in this submission except interface *PolicyEvaluatorLocatorPatternAdmin*. In this case attribute *pattern_admin* of *PolicyEvaluatorLocator* interface implementation must have value *null*.

An implementation of Resource Access Decision facility compliant to conformance class **“RAD with Patterns**” must implement all of the interfaces defined in this submission. In this case *pattern_admin* data member of *PolicyEvaluatorLocator* interface implementation must have non *null* value.

1.5 Complete IDL

```
//File: DfResourceAccessDecision.idl
//

#ifndef _DF_RESOURCE_ACCESS_DECISION_IDL_
#define _DF_RESOURCE_ACCESS_DECISION_IDL_

#include “Security.idl”
#pragma prefix “omg.org”

module DfResourceAccessDecision {


    //*********************************************************
    //        Basic Types
    //*********************************************************
    typedef sequence<boolean> BooleanList;
    typedef Security::AttributeList AttributeList;

    interface DynamicAttributeService;
    interface DecisionCombinator;
    interface PolicyEvaluator;
    interface PolicyEvaluatorLocator;
    interface PolicyEvaluatorLocatorAdmin;
    interface PolicyEvaluatorAdmin;

    //*********************************************************
    //   Types that identify a secured resource
    //*********************************************************
    struct ResourceNameComponent {
        string name_string;
        string value_string;
    };
    typedef sequence<ResourceNameComponent>
        ResourceNameComponentList;

    typedef string ResourceNamingAuthority;

    struct ResourceName {
        ResourceNamingAuthority resource_naming_authority;
        ResourceNameComponentList resource_name_component_list;
    };

    typedef ResourceNameResourceNamePattern;

    typedef string Operation;
    typedef sequence<Operation> OperationList;
```

```
//*****************************************************
//   Types associated with evaluating Access Policy
//*****************************************************
typedef string PolicyName;
typedef sequence<PolicyName> PolicyNameList;

const PolicyName NO_ACCESS_POLICY = “NO_ACCESS_POLICY”;

struct NamedPolicyEvaluator {
     string evaluator_name;
     PolicyEvaluator policy_evaluator;
};
typedef sequence<NamedPolicyEvaluator> PolicyEvaluatorList;

struct PolicyDecisionEvaluators {
     PolicyEvaluatorList policy_evaluator_list;
     DecisionCombinator decision_combinator;
};


//*****************************************************
//      Types used to request an Access Decision
//*****************************************************

struct AccessDefinition {
     ResourceName resource_name;
     Operation operation;
};
typedef sequence<AccessDefinition> AccessDefinitionList;

enum DecisionResult {
     ACCESS_DECISION_ALLOWED,
     ACCESS_DECISION_NOT_ALLOWED,
     ACCESS_DECISION_UNKNOWN
};


//*********************************************************
//           Exception Data types
//*********************************************************


struct ExceptionData {
     short error_code;
     string reason;
};
enum InternalErrorType {Fatal, NotFatal};


//*********************************************************
//    Exception thrown by the Access Decision Object
```

```
//*********************************************************

exception InternalError{InternalErrorType ed;};


//*********************************************************
//    Exception thrown by Internal non-admin interfaces
//*********************************************************

exception ComponentError{
     ExceptionData ed;
     InternalErrorType it;
};


//*********************************************************
//        Exceptions thrown by Admin Interfaces
//*********************************************************

exception PatternConflict {ExceptionData ed;};
exception PatternDuplicate {ExceptionData ed;};
exception PatternNotRegistered {ExceptionData ed;};
exception PatternInUse {ExceptionData ed;};
exception InputFormatError {ExceptionData ed;};
exception ResourceNameNotFound {ExceptionData ed;};
exception NoAssociation {ExceptionData ed;};
exception InvalidPolicy {ExceptionData ed;};
exception DuplicateEvaluatorName {ExceptionData ed;};
exception InvalidResourceName {};
exception InvalidResourceNamePattern {};

exception InvalidPolicyEvaluatorList {
     ExceptionData ed;
     NamedPolicyEvaluator first_invalid_element;
};

exception InvalidPolicyNameList {
     ExceptionData ed;
     PolicyName first_invalid_element;
};

//*****************************************************
//     interface AccessDecision
//*****************************************************

interface AccessDecision {

     boolean access_allowed(
          in  ResourceName          resource_name,
          in  Operation             operation,
          in  AttributeList         attribute_list
```

```
        )
        raises (InternalError);

        BooleanList multiple_access_allowed(
            in  AccessDefinitionList    access_requests,
            in  AttributeList           attribute_list
        )
        raises (InternalError);

};


//*******************************************************
//    interface DynamicAttributeService
//*******************************************************

interface DynamicAttributeService {

        AttributeList get_dynamic_attributes(
            in  AttributeList           attribute_list,
            in  ResourceName            resource_name,
            in  Operation               operation
        )
        raises (ComponentError);

};

//*******************************************************
//    interface PolicyEvaluatorLocator
//*******************************************************

interface PolicyEvaluatorLocator {

        readonly attribute PolicyEvaluatorLocatorBasicAdmin basic_admin;
        readonly attribute PolicyEvaluatorLocatorNameAdmin name_admin;



        readonly attribute PolicyEvaluatorLocatorPatternAdmin
        pattern_admin;

        PolicyDecisionEvaluators get_policy_decision_evaluators(
            in  ResourceName            resource_name
        )
        raises (ComponentError);

};


//*******************************************************
//    interface DecisionCombinator
```

```
    //*****************************************************

    interface DecisionCombinator{

        boolean combine_decisions(
            in  ResourceName        resource_name,
            in  Operation           operation,
            in  AttributeList       attribute_list,
            in  PolicyEvaluatorList policy_evaluator_list
        )
        raises (ComponentError);
};


    //*****************************************************
    //    interface PolicyEvaluator
    //*****************************************************

    interface PolicyEvaluator {

        readonly attribute PolicyEvaluatorAdmin pe_admin;

        DecisionResult evaluate(
            in  ResourceName        resource_name,
            in  Operation           operation,
            in  AttributeList       attribute_list
        )
        raises (ComponentError);

    };

    //*****************************************************
    //
    //   Management Interfaces
    //
    //*****************************************************

    //*****************************************************
    //    interface AccessDecisionAdmin
    //*****************************************************

    interface AccessDecisionAdmin {

        PolicyEvaluatorLocator get_policy_evaluator_locator();

        void set_policy_evaluator_locator (
            in  PolicyEvaluatorLocator policy_evaluator_locator
        );

        DynamicAttributeService get_dynamic_attribute_service();
```

```
        void set_dynamic_attribute_service(
            in  DynamicAttributeService dynamic_attribute_service
        );
};


//*******************************************************
//    interface PolicyEvaluatorLocatorBasicAdmin
//*******************************************************

interface PolicyEvaluatorLocatorBasicAdmin {

        PolicyEvaluatorList set_default_evaluators(
            in  PolicyEvaluatorList policy_evaluator_list
        )
        raises (DuplicateEvaluatorName, InvalidPolicyEvaluatorList);

        PolicyEvaluatorList get_default_evaluators();

        DecisionCombinator get_default_combinator ();

        void set_default_combinator (
            in  DecisionCombinator decision_combinator
        );
}


//*******************************************************
//    interface PolicyEvaluatorLocatorNameAdmin
//*******************************************************

interface PolicyEvaluatorLocatorAdmin {

        PolicyEvaluatorList get_evaluators(
            in  ResourceName          resource_name
        )
        raises (InvalidResourceName);

        void set_evaluators (
            in  PolicyEvaluatorList     policy_evaluator_list,
            in  ResourceName            resource_name
        )
        raises (
            InvalidPolicyEvaluatorList,
            InvalidResourceName,
            DuplicateEvaluatorName
        );

        void add_evaluators (
            in  PolicyEvaluatorList     policy_evaluator_list,
            in  ResourceName            resource_name
```

```
        )
        raises (
            InvalidResourceName,
            InvalidPolicyEvaluatorList,
            DuplicateEvaluatorName
        );

        void delete_evaluators (
            in  PolicyEvaluatorList     policy_evaluator_list,
            in  ResourceName            resource_name
        )
        raises (
            InvalidResourceName,
            InvalidPolicyEvaluatorList,
            DuplicateEvaluatorName
        );

        DecisionCombinator get_combinator (
            in  ResourceName            resource_name
        )
        raises (InvalidResourceName);

        void set_combinator (
            in  DecisionCombinator      decision_combinator,
            in  ResourceName            resource_name
        )
        raises (InvalidResourceName);

        void delete_combinator (
            in  ResourceName            resource_name
        )
        raises (InvalidResourceName);
    };


    //*********************************************************
    //    interface PolicyEvaluatorLocatorPatternAdmin
    //*********************************************************

    interface PolicyEvaluatorLocatorPatternAdmin {

        voidregister_resource_name_pattern(
            in  ResourceNamePattern  pattern
        )
        raises (
            InvalidResourceNamePattern,
            PatternDuplicate,
            PatternConflict
        );

        voidunregister_resource_name_pattern(
```

```
        in  ResourceNamePattern  pattern
    )
    raises (
        InvalidResourceNamePattern,
        PatternNotRegistered,
        PatternInUse
    );

    PolicyEvaluatorList get_evaluators_by_pattern(
        in ResourceNamePattern   pattern
    )
    raises (
        InvalidResourceNamePattern,
        PatternNotRegistered
    );

    voidset_evaluators_by_pattern (
        in  PolicyEvaluatorList      policy_evaluator_list,
        in  ResourceNamePattern  pattern
    )
    raises (
        InvalidPolicyEvaluatorList,
        InputFormatError,
        PatternNotRegistered,
        DuplicateEvaluatorName
    );

    PolicyEvaluatorList set_default_evaluators(
        in  PolicyEvaluatorList      policy_evaluator_list
    )
    raises (
        DuplicateEvaluatorName,
        InvalidPolicyEvaluatorList
    );

    voidadd_evaluators_by_pattern (
        in  PolicyEvaluatorList      policy_evaluator_list,
        in  ResourceNamePattern  pattern
    )
    raises (
        InvalidResourceNamePattern,
        PatternNotRegistered,
        InvalidPolicyEvaluatorList,
        DuplicateEvaluatorName
        );

    voiddelete_evaluators_by_pattern (
        in  PolicyEvaluatorList      policy_evaluator_list,
        in  ResourceNamePattern  pattern
    )
    raises (
```

```
            InvalidResourceNamePattern,
            PatternNotRegistered,
            InvalidPolicyEvaluatorList,
            DuplicateEvaluatorName
        );

        DecisionCombinatorget_combinator_by_pattern (
            in  ResourceNamePattern  pattern
        )
        raises (
            InvalidResourceNamePattern,
            PatternNotRegistered
        );

        voidset_combinator_by_pattern (
            in  DecisionCombinator    decision_combinator,
            in  ResourceNamePattern  pattern
        )
        raises (
            InvalidResourceNamePattern,
            PatternNotRegistered
            );

        voiddelete_combinator_by_pattern (
            in  ResourceNamePattern  pattern
        )
        raises (
            InvalidResourceNamePattern,
            PatternNotRegistered
        );

        DecisionCombinator get_default_combinator ();

        void set_default_combinator(
            in  DecisionCombinator    decision_combinator
        );
    };


    //*********************************************************
    //    interface PolicyEvaluatorAdmin
    //*********************************************************

    interface PolicyEvaluatorAdmin {

        void set_policies(
            in  PolicyNameList          policy_names,
            in  ResourceName            resource_name
        )
        raises (
            InvalidResourceName,
```

```
            ResourceNameNotFound,
            InvalidPolicyNameList
        );

        void add_policies(
            in  PolicyNameList          policy_names,
            in  ResourceName            resource_name
        )
        raises (
            InvalidResourceName,
            ResourceNameNotFound,
            InvalidPolicyNameList
        );

        void delete_policies(
            in  PolicyNameList          policy_names,
            in  ResourceName            resource_name
        )
        raises (
            InvalidResourceName,
            ResourceNameNotFound,
            InvalidPolicyNameList,
            NoAssociation
        );

        PolicyNameList list_policies();

        PolicyName set_default_policy(
            in  PolicyName              policy_names
        )
        raises (InvalidPolicy);
    };

};

#endif  // _DF_RESOURCE_ACCESS_DECISION_IDL_
```