FLORIDA INTERNATIONAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE

THE CENTER FOR ADVANCED DISTRIBUTED SYSTEMS ENGINEERING (CADSE)

TECHNICAL REPORT

2000-01

DESIGN AND IMPLEMENTATION OF RESOURCE ACCESS DECISION SERVER

Luis Espinal, Konstantin Beznosov, Yi Deng
{lespin03,beznosov,deng}@cs.fiu.edu

January 18, 2000

Abstract

Decoupling authorization decision logic enables implementation of complex and consistent access control policies across heterogeneous systems. However, this is difficult, if not impossible to implement by exclusively using general-purpose infrastructures such as CORBA Security Service. In response to this limitation of CORBA Security service the Object Management Group (OMG) has adopted a Resource Access Decision (RAD) Facility, an authorization service for distributed systems, as a pre-final standard. By using RAD facility, developers can implement systems with authorization logic decoupled from application-specific logic and decentralized evaluation and administration of the access policies.

This report documents the design and implementation of a Resource Access Decision (RAD) facility. The report covers the different components that comprise a RAD system, their designs, functions and interdependencies. The RAD prototype allows studying the validity of the framework and conduction of experiments in the research of distributed access control. Since the design of the prototype is heavily influenced by design patterns, the prototype can easily be maintained and augmented with more complex access control mechanisms.

Keywords

Authorization, access control, resource access decision, CORBA, Java, security, authorization, software engineering, distributed security, distributed systems.

Design and Implementation of Resource Access Decision Server

Luis Espinal, Konstantin Beznosov, Yi Deng
The Center for Advanced Distributed Systems Engineering (CADSE)
Florida International University, Miami, FL
{lespin03,beznosov,deng}@cs.fiu.edu

1. INTRODUCTION

Contemporary enterprise systems have increased in size and complexity and tend to be distributed across organizations with potentially heterogeneous computing platforms. Organizations in need of such computing infrastructures rely on existing software middleware such as OMG's Common Object Request Broker Architecture (CORBA) to build such systems. CORBA defines a platform and language independent object-oriented middleware that allows integration of diverse systems [17]. This integration permits transparent access to remote services [17][24]. CORBA in general, and CORBA Security in particular, provides uniform, general-purpose infrastructure with which to build secured object-oriented distributed systems for a wide variety of application domains. In order to build such systems, however, developers need to extend this infrastructure and implement all the necessary features required for a particular system. This is especially true for applications that require complex application domain-specific authorization decisions [20]. Moreover, developers need to have an architectural view of the system under development. By architectural view we mean the structures constituting the components of the system, the nature and role of each component, their interfaces and expected interactions and system-wide properties or semantics [14]. Unfortunately, these factors are beyond the intended scope of CORBA general-purpose infrastructure.

As a result, the need to realize these requirements and the lack of general approaches to accomplish them has lead developers to thightly couple domain-specific authorization and application-specific logic. By application-specific logic we mean business domain logic which is separate of architectural factors or requirements such as security, transaction, performance, availability or scalability to name a few. Although coupling application-specific logic and authorization logic (or other architectural concerns for that matter), allows developers to realize the intended systems, the separation of the two gives significant benefits. Systems resulting from this separation are easier to manage since there is a clear separation of responsibilities between security administrators and developers [12].

In response to this limitation of CORBA Security service, at the time this report is written (2000), a Resource Access Decision (RAD) Facility [12][20] has been adopted by the Object Management Group (OMG) as a pre-final standard. RAD facility (or server as it is also referred in this report) provides mechanisms to obtain authorization decisions. By using RAD facility, developers can implement systems with authorization logic decoupled from application-specific logic and decentralized evaluation and administration of the access policies. Another advantage of using RAD facility is that it partitions system changes that can (and will) occur into application-specific and authorization decision changes. That is, changes in authorization decision logic are contained within the RAD facility without having great impact in application-specific logic and vice versa. The RAD facility complements CORBA Security access model, and allow developers to implement access control mechanisms of arbitrary granularity [12].

As part of research at the Center for Advanced Distributed Systems Engineering (CADSE), a prototype of the RAD server has been implemented. The prototype allows studying the validity of such a framework and conduction of various experiments in the research of distributed access control. The prototype is also used for reasoning about properties and semantics of the prototype itself, other implementations of the RAD server, and of applications using the prototype. The prototype can serve as a vehicle to study approaches for implementing extensible, maintainable solutions for authorization decision problems in a

cost-effective manner. This report documents the design decisions made during the implementation of the prototype and the forces influencing its design and implementation.

The organization of this report is the following: Section 2 gives an overview of the RAD specification including a brief overview of CORBA and scope of the authorization service. Section 3 introduces the architecture of the prototype, its components, their functions, interfaces and interactions and observable properties or semantics of the components. Lastly, section 4 describes the implementation of the prototype, the design patterns used in the implementation of the components, and the additional implementation-specific properties of the component.

2. Overview of RAD Specification

The function of a RAD server is to administer and enforce security policies of varying complexity. As it is defined in [20], a RAD server is built using CORBA; nevertheless, the same ideas could be implemented using other middleware technologies such as DCOM or SunRPC. Notice that there is a RAD specification and a RAD server. The RAD specification stands for the requirements, specifications, interfaces and suggestions that guide the implementation of a CORBA facility as specified in [20] in response to the HRAC RFP [19]. A RAD server is a concrete, executable implementation compliant with the RAD facility. Throughout this report, unless specified otherwise, the term "RAD server" denotes the RAD prototype implemented at CADSE.

2.1. CORBA Overview

CORBA defines the programming interfaces and middleware architecture with which to develop object-oriented distributed systems. The building blocks of a CORBA-based system are CORBA objects, and processes executing programs that contain CORBA objects are referred to as CORBA servers (or simply servers) [5]. In a CORBA-based system, a CORBA object makes its services available to other potentially distributed CORBA objects. This involves giving CORBA objects a representation which can be manipulated and managed by other CORBA objects [15]. To that end, services provided by a CORBA object are defined through interfaces written in OMG Interface Definition Language (IDL). Clients using the operations provided by a CORBA object only know about its interface and need not worry about the implementation and locality of the CORBA object. That is, a client treats a CORBA object the same way it treats objects in the client address space.[1]

CORBA interfaces can be categorized as CORBA services and CORBA facilities [19]. CORBA services are general purpose services fundamental to the construction of CORBA-based systems or universal, domain-independent services. CORBA facilities are also general interfaces applicable to most domains. The difference is that CORBA facilities are end-user oriented in nature. In [19] and [20], RAD is specified as a CORBA facility.

2.2. Scope of Authorization Service

A RAD server implements mechanisms for obtaining authorization decisions [20]. To understand what we mean by "authorization decision", it is necessary to introduce several concepts pertaining to access (security) policies and access control. An *access policy* defines the security requirements of a system. These security requirements govern how and when *principals* (users or systems running on behalf of users) operate and access system resources [6]. These access policies are enforced by means of *access control* mechanisms which grant or deny principals' requests for access to resources [6].

1. Some authors argue that distributed objects such as CORBA objects cannot be treated as objects located in a single address space since issues such as latency and partial failure are intrinsic observable properties of their interfaces [11].

Access control mechanisms enforce (*mediate*) grant or denial of requests by subjects (or *principals*) for access to secured resources. During mediation, access control mechanisms *evaluate* applicable access policies, and the results from these evaluations determine whether to grant or deny a request. The decision to grant or deny access to a resource based on an access policy evaluation is known as an *authorization decision*.

In a traditional scenario, the application server implements both the access control and authorization decision logic (Figure 1). With RAD, the authorization decision logic is moved to an authorization server, and the application server is expected to enforce the authorization decision (Figure 2). Thus, usage of the RAD server allows decoupling of authorization logic from application logic. Also, the RAD server is used to provide a standard interface to security-aware clients for requesting access control decisions [20].

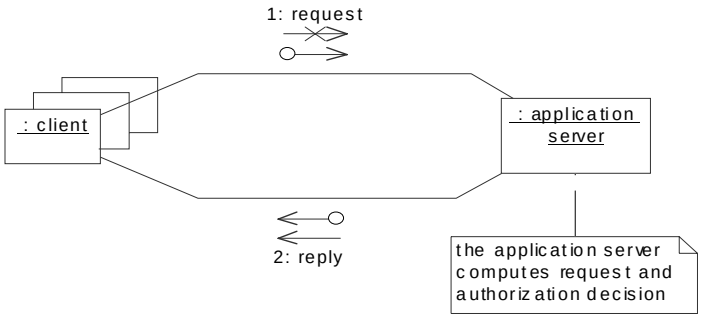

Figure 1: Authorization logic implemented in the application server

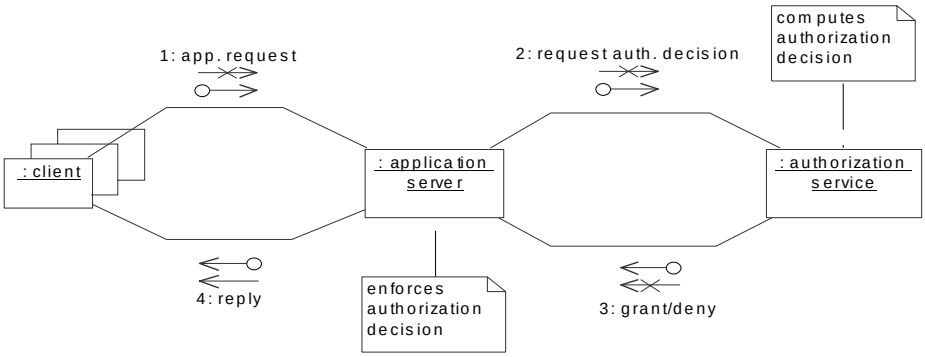

Figure 2: Authorization logic residing in RAD (authorization decision) server

In RAD, the reply sent to the application server can either be the authorization decision (grant/deny) or an exception indicating an internal failure while computing an authorization decision. This stresses the role of the RAD as an access decision server (not as an access control server). As a consequence, the application server must not only enforce the authorization decisions, but must also make the policy enforcement decisions about how to proceed during exceptional circumstances [20]. It is important to understand that the RAD server is concerned with implementing authorization decision logic at the application level. Issues such as access control at the operating system level, authentication and intrusion detection falls outside the current scope of the RAD server.

3. ARCHITECTURE OF RAD SERVER

In this report, the architecture of the RAD server describes the components that make up the RAD server, their externally visible properties and the interaction among them [14]. By component we mean a replaceable unit of computation that provides services or operations to other services and may use opera-

tions on other components. By externally visible properties we mean the assumptions other components can make about the component such as performance characteristics and services provided. This definition also includes behavior discernible from the point of view of another component [14].

3.1. Components of RAD Server

A RAD server is composed of the following components [12][20]:

1. AccessDecision Object (ADO)
2. PolicyEvaluatorLocator (PEL)
3. DynamicAttributeService (DAS)
4. DecisionCombinator (DC)
5. PolicyEvaluator (PE).

Application servers (clients from the RAD server point of view) interact with RAD server only through the ADO.[1] That is, the ADO acts as a facade that provides a single, uniform interface to the other interfaces that make the RAD server [8]. Given a resource name (a secured resource identifier), there can be zero or more access control policies governing access to it. The evaluation of such policies is done by the Policy Evaluator (PE) objects.

After evaluation of a policy, a PE returns a grant or deny access (yes/no) or "don't know" answer. A "don't know" answer is used by a PE when it cannot perform an evaluation. Also, A PE object can evaluate one or more access policies for a given resource. However, access policies associated with a resource are not necessarily evaluated by a single PE object. Therefore, we have a one-to-many relation from PE objects to access policies and a many-to-many relation from access policies to resources (Figure 3).

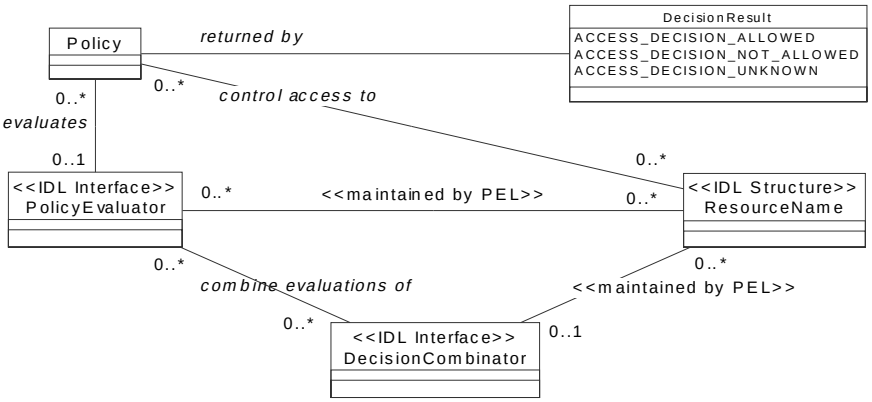

Figure 3: Relationship among PE's, access policies, resource names and DC's

Since there can be more than one access policy for a resource name, RAD uses a Decision Combinator object to combine all policy evaluations into a single grant/deny authorization decision which is sent back to the client. A DC objects provides a *combination policy.* For example, a DC can implement an *open world* or *closed world* policy.[2] More complex decision combination policies can be implemented such as granting access based on a majority vote or on hierarchies of PE objects where a decision from a higher-

1. The RAD ADO has different semantics from the CORBAsec object of the same name even though they have similar functions [20].

2. With open world combination strategy, a DC returns yes (grant access) only if none of the PE returns a no (deny). A closed world combination strategy grant access only if all PE's grant access. In essence, open world strategy grants access unless there is an explicit denial from a PE, and closed world strategy grants access only explicit grant access from all PE's.

level PE can override decisions from lower-level PE objects. When a PE cannot perform its evaluations and returns “don’t know”, the combination policy used by DC will determine whether it returns “grant” or “deny”.

Locating PE and DC Components

Whenever the ADO receives an authorization decision request, it needs to know what DC and PE(s) are applicable to the given secured resource. To this end, the ADO uses the PolicyEvaluatorLocator (PEL) object which decides what DC and PE(s) to use. A PEL maintains the relations of DC’s, PE’s and resource names, and isolates the ADO from potentially complex mechanisms for resolving and administering these relations. Resolving mechanisms can range from a simple search in a local database to resolving references to remote objects. Administrative mechanisms can range from setting default DC and PEs for all secured resources to associating these objects to PE objects to group of secured resources matched by a resource name pattern (please see Section 4.3).

Dynamic Attributes

To evaluate an access policy, a PE needs to know what are the resource name, the intended operation on the resource, and the characteristics of the principals or *security attributes*. These security attributes are used by PE objects as criteria for evaluating access control policies. It is based on these security attributes that an authorization decision is made.[1] The principal’s security attributes can contain both static and dynamic attributes.[2] Static attributes represent the characteristics of the principal set by an administrator (e.g. user name and role) which do not change while a principal operates in the system [20].

On the other hand, a dynamic attribute can only be determined at the time an access request takes place. These dynamic attributes can denote relationships between a principal and a resource [20]. For example, a physician can access a patient’s medical records only if he is the attending physician for the patient. This “attending physician” relationship between physicians and patients is subject to change from one authorization request to another. In this scenario, the ADO delegates the discovery of dynamic attributes to a Dynamic Attribute server (DAS). The DAS itself can become a proxy [7][8] to other, more specialized dynamic attribute servers or SDAS.

Control Flow

In summary, an authorization decision is computed through a sequence of operations carried out by the RAD components (Figure 4). The following algorithm describes how RAD components compute an authorization decision:

Algorithm 1 Computing One Authorization Request

1) An application server (AS for short) contacts the ADO server for an authorization decision to perform an operation P on resource R by a principal U with a list (or set) of security attributes $\{a\}$.

2) The ADO object requests the PEL object for references to a DC and any PE servers associated to the resource R.

3) The PEL returns to the ADO a reference to a DC and a set $\{pe\}$ with zero or more references to PE objects. $\{pe\}$ represents the PE objects associated with resource R at the time the request for authorization decision takes place.

4) The ADO requests the DAS for any dynamic attributes of U with respect to R and P at the time the request for the authorization decision takes place.

1. The authorization decision obtained from a policy evaluation can also be determined from implicit parameters such as the time the request is made. How this implicit parameters are handled or configured are outside the scope of this report.
2. For more information on security attributes, operations and resource names and their definitions, please refer to [12][18] and [20]

5) The DAS returns to the ADO a set $\{a'\}$ to be used in obtaining an authorization decision. Notice that the contents of $\{a'\}$ depend on the configuration of the DAS as well as of what dynamic attributes are available at the time the request takes place. The DAS can add dynamic attributes or remove existing attributes from set $\{a\}$.

6) The ADO sends to the DC a set of PE servers $\{pe\}$ for evaluation of policies that control access to the resource R.

7) The DC requests each PE in $\{pe\}$ to authorize or deny the operation P on the resource R given the security attributes $\{a'\}$ of the principal.

7.1) Each PE in $\{pe\}$ evaluates zero or more access policies associated with resource R. Depending on the implementation of the PE server, it will combine all these policies into a single YES/NO/ DON'T KNOW reply. This reply is then returned to the DC server.

8) The DC combines all replies from all the PE servers in $\{pe\}$, and combines them into a single grant or deny response. This response, the authorization decision, is returned to the ADO server.

9) The ADO returns the authorization decision from the DC server to the AS server.

10) The AS server receives the authorization decision from the ADO server and enforces it. The manner in which the authorization decision is enforced depends on the implementation of the AS server.

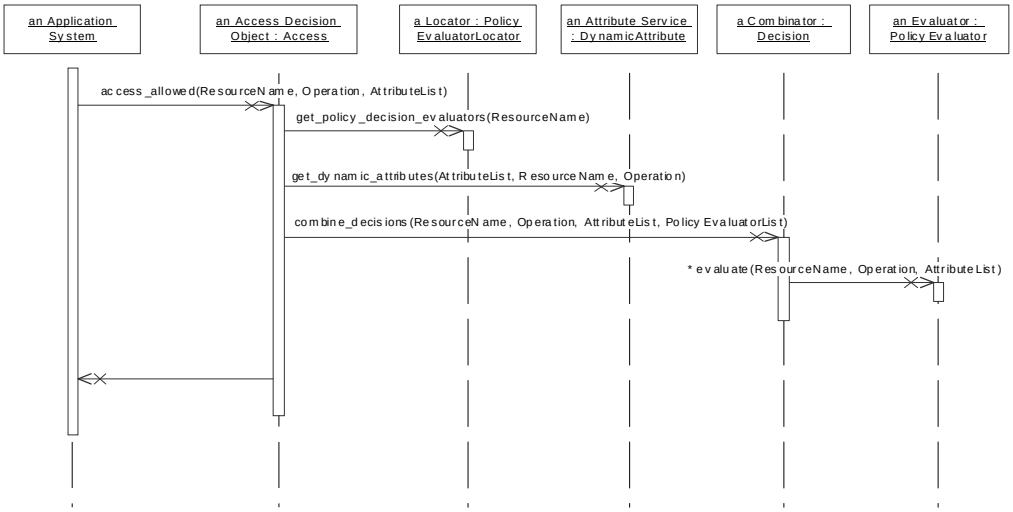

Figure 4: Sequence Diagram - 1 Authorization Request

3.2. Computational Model

The RAD specification [20] introduces several standard interfaces that can be classified as run-time and administrative interfaces. The run-time interfaces describe the objects and operations a client uses to obtain an access decision from the RAD server. The administrative interfaces, on the other hand, describe the objects and operations involved in the configuration of the RAD server.

The implementation of the prototype RAD server introduces extensions to the required interfaces (see Figure 5). The extensions to the run-time interfaces provide operations to get references to administrative interfaces. Extensions to the administrative interfaces provide operations for gracefully shutting down components. Table 1 lists the standard and extended interfaces, and table 2 lists the classes implementing the extended interfaces and the packages in which the classes are organized.[1]

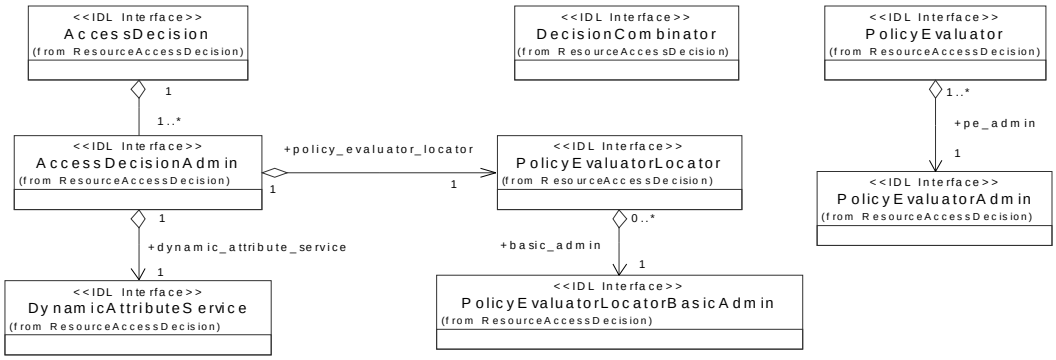

Figure 5: RAD Server Computational Model - Implemented Interfaces only.

RAD Component	Standard IDL Interface	Extended IDL Interface
ADO	AccessDecision	AccessDecisionExt
	AccessDecisionAdmin	AccessDecisionAdminExt
PEL	PolicyEvaluatorLocator	
	PolicyEvaluatorLocatorAdmin	
	PolicyEvaluatorLocatorBasicAdmin	PolicyEvaluatorLocatorAdminExt
	PolicyEvaluatorLocatorNameAdmin	
	PolicyEvaluatorLocatorPatternAdmin	
DAS	DynamicAttributeService	DynamicAttributeServiceExt
		DynamicAttributeServiceAdminExt
DC	DecisionCombinator	
PE	PolicyEvaluator	PolicyEvaluatorExt
	PolicyEvaluatorAdmin	PolicyEvaluatorAdminExt

Table 1: Implemented interfaces

RAD Component	Java Package
ADO	edu.fiu.cadse.rad.ado
PEL	edu.fiu.cadse.rad.pel
DAS	edu.fiu.cadse.rad.das
DC	edu.fiu.cadse.rad.dc
PE	edu.fiu.cadse.rad.pe

Table 2: Implemented packages

4. IMPLEMENTATION

Although the RAD components have different functions, their implementation faced two main problems. The first problem was to devise an initialization mechanism common to all RAD components. The second problem was to implement IDL interfaces using a single class even though Java does not support multiple inheritance.

1. At the time this report is written (2000), the prototype did not have an implementation for the PolicyEvaluatorLocatorNameAdmin and PolicyEvaluatorLocatorPatternAdmin, nor did it introduce extension interfaces for the DC.

4.1. Component Implementation and Initialization

A RAD component can have multiple interfaces (run-time and administrative). Since each administrative interface is used to configure the run-time interface of a component (which may require some degree of interdependency), it was be desirable that each component has a single class implementing both the run-time and administrative interfaces. Another desirable property was to have a generic approach to initialize, or bootstrap the components.

This initialization mechanism should be transparent to the component. The meaning of this requirement is twofold. First, changes to the initialization logic should not introduce changes in the implementation of the components. Likewise, a component should not expect a particular initialization procedure insofar as this initialization provides to the component a working environment on which it can interact with other components as expected.

Implementation of Multiple Interfaces

In Java, an IDL interface is implemented using a class which provides the minimum mechanisms needed to interact with the ORB environment and defines public methods corresponding to the operations and attributes of the IDL interface [16]. However, since Java does not support inheritance of multiple classes, inheritance could not be used for implementing the run-time and administrative IDL interfaces.

To work around the single-inheritance restriction of Java, components were implemented using a delegation mechanism known as *Tie Approach* [5]. In the tie approach, a *tie* class implements a given CORBA interface or interfaces.[1] However, the *tie* only implements the minimum mechanisms needed to interact with the ORB environment. The actual implementation of the component's operations is done in a *delegate* class implementing the *ComponentOperation* interface. Figure 6 illustrates an IDL interface being implemented using *tie* objects. With the tie approach we obtain greater flexibility in composing objects since the delegate class is not restricted to inherit from any particular class. The only requirement is that the delegate class implements the *ComponentOperation* interface.[2]

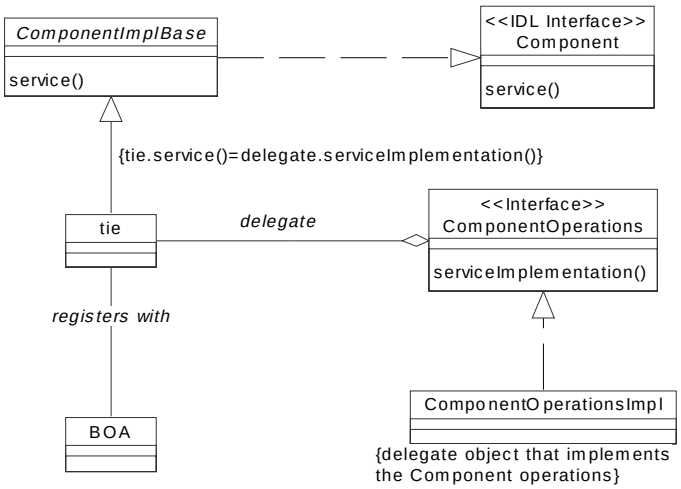

Figure 6: Implementing a CORBA object using the tie approach

Object Composition

During design, it became evident that RAD components such as DC and PE can exhibit different behavior. For instance, a DC can combine results from multiple PEs in more than one way. One solution would

1. Tie approach in the CORBA community means the use of delegation over inheritance when implementing IDL interfaces [5]. In principle, this is similar to the *Bridge* pattern described in [7][8].
2. One drawback of delegation is that systems that rely on object composition may be more difficult to comprehend [8].

be to implement one class per component behavior. However, this would create many related classes that differ only in their behavior. The solution we chose was based on the *Strategy* pattern.

In *Strategy* pattern, a *Context* class implements the logic common to all other implementations of a base class (a RAD component in our case), and a *Strategy* class (interface in our case) provides behavior specific to an implementation (Figure 7). *Strategy* pattern allowed us to implement families of algorithms related to each RAD component (strategy classes) and common functionality (context) classes [7][8][13].

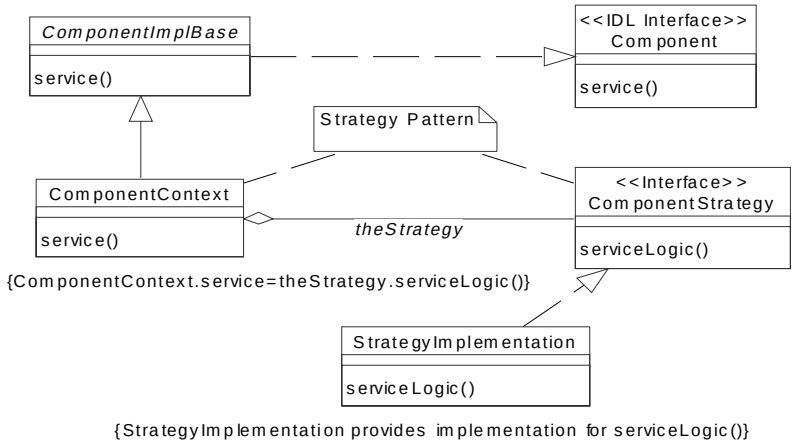

Figure 7: Implementing a server using *Strategy* pattern

In the implementation of the RAD prototype, we define strategies as Java interfaces. In this case, component contexts are Java classes implementing the services published by the strategy interfaces. With the implementation of the strategies for the DC and PE components, we took a step further: their implementation is based on a design pattern known as *Template* (Figure 8). The idea behind *Template* pattern (or *Tem-*

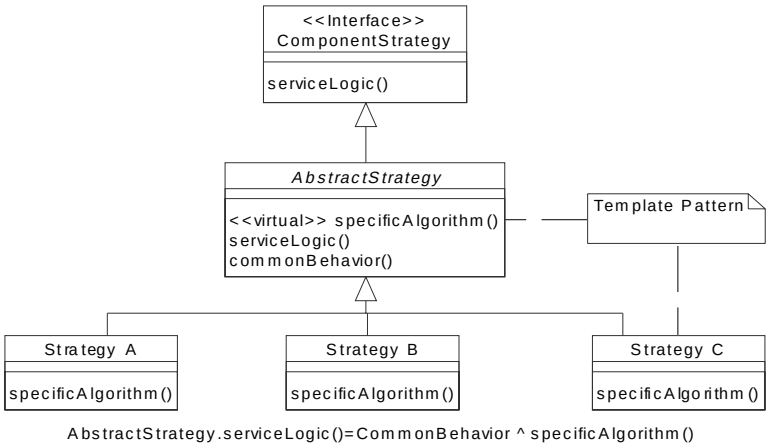

Figure 8: Extending an interface using template pattern

plate Method pattern) is to define an outline or skeleton of an algorithm in a base class while leaving some steps to be defined in subclasses [7][8][13].

Template pattern was used in the design of DC and PE because implementations of these components tend to share common functionality. For example, implementations of DC need to resolve references to PE objects received from the ADO regardless of the decision combination policy being implemented. Similarly, implementations of PE need to maintain associations of policies to resource names independently of

how the policies are maintained and evaluated. Such common functionality or behavior can be implemented in an abstract strategy class (Figure 8). This abstract strategy class can later be extended or refined to obtain specific implementations of DC and PE.

Run-time and Administrative Interfaces

Each RAD component (with the exception of DC) has a run-time and an administrative interface. Figure 9 illustrates our approach to implement both interfaces. The implementations of ADO, DAS and PE follow such an approach. PolicyEvaluatorLocator (PEL) does not have an extension to its administrative interface as it will be explained with more details in Section 4.4.3. The DecisionCombinator, on the other hand does not have an administrative interface due to its simplicity (see Section 4.4.5.)

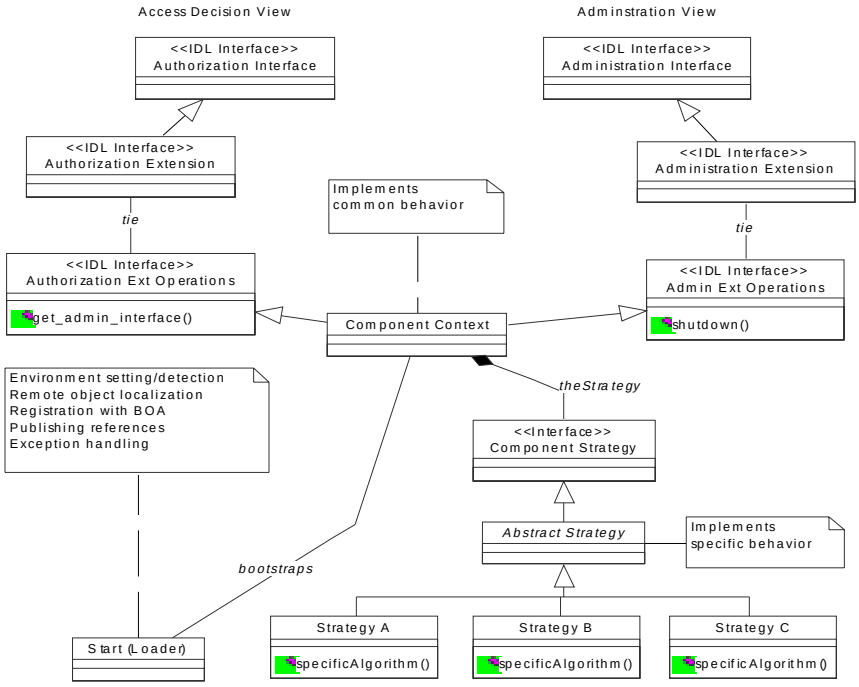

Figure 9: General approach for implementing a RAD component

Component Initialization

Each RAD component in our implementation depends on a *Start* class for its initialization. The required environment for each component may differ from one to the other; however, all start classes follow a similar initialization process. This process includes resolving references to remote CORBA objects, creation of objects, vendor-dependent ORB detection, registration of interfaces with BOA, and exception handling. Figure 9 illustrates our general approach to implement the run-time and administrative interfaces and initialize or bootstrap the component or server.

4.2. AccessDecision Object

The run-time and administrative interfaces of the *AccessDecision* Object (ADO) are represented by the *AccessDecision* and *AccessDecisionAdmin* interfaces [20]. Clients communicate with the RAD facility through the *AccessDecision* interface. Meanwhile, the function of the *AccessDecisionAdmin* interface is to provide a means to configure the ADO object. Configuring the ADO object means setting its references to PEL and DAS objects. Its extension, the *AccessDecisionAdminExt* also provides a means for shutting down the ADO object. The extension to the *AccessDecision* interface, *AccessDecisionExt* provides mechanism to obtain a reference to the administrative interface.

As will be described in section 4.4.1, we used a single Java object to implement the *AccessDecisionExt* and *AccessDecisionAdminExt*, the *ResourceAccessDecider* class in edu.fiu.cadse.rad.ado package (see Figure 10).

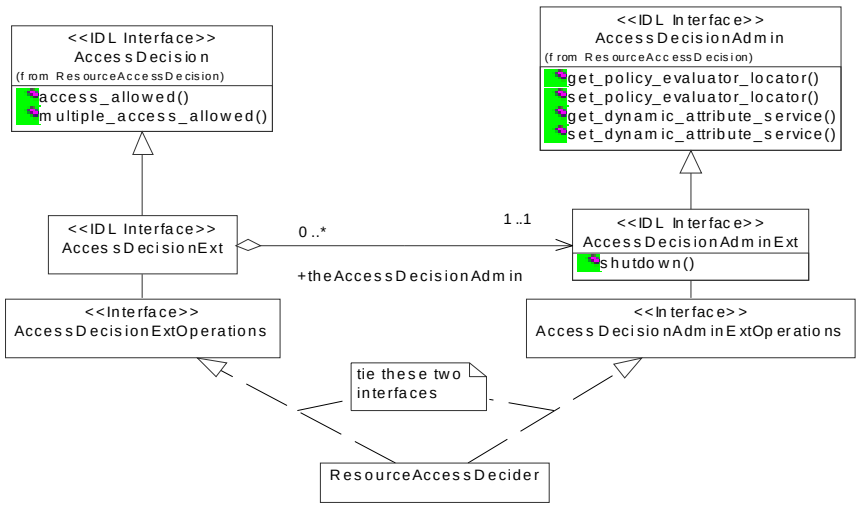

Figure 10: Implementation of the AccessDecision object

4.3. PolicyEvaluatorLocator

PolicyEvaluatorLocator (PEL) is represented by *PolicyEvaluatorLocator* (run-time) and *PolicyEvaluatorLocatorAdmin* (administrative) interfaces. The function of the PEL object is to return a *DecisionCombinator* (DC) and a list of *PolicyEvaluator* (PE) objects for authorization decisions on a resource name [12][20]. In principle, a PEL could return 1) default DC and PE objects for all resource names, 2) DC and PE objects specifically associated with a given resource name, or 3) DC and PE objects associated with a family of resource names which can be matched with a resource name pattern (please see Section).

The RAD specification [20] introduces three administrative interfaces for the PEL component: *PolicyEvaluatorLocatorBasicAdmin*, *PolicyEvaluatorLocatorNameAdmin* and *PolicyEvaluatorLocatorPatternAdmin*. The *PolicyEvaluatorLocatorBasicAdmin* is used to administer default associations between PolicyEvaluators (or DecisionCombinators) and resource names [20]. The *PolicyEvaluatorLocatorNameAdmin* is used to set explicit associations between DC and PE objects and resource names. These associations take precedence over default associations set through *PolicyEvaluatorLocatorBasicAdmin*.

The *PolicyEvaluatorLocatorPatternAdmin* is used to administer associations based on resource name patterns. As with *PolicyEvaluatorLocatorNameAdmin*, associations set by *PolicyEvaluatorLocatorPatternAdmin* take precedence over default associations set through *PolicyEvaluatorLocatorBasicAdmin*. The prototype of the RAD server does not include the *PolicyEvaluatorLocatorNameAdmin* and *PolicyEvaluatorLocatorPatternAdmin*, yet. Please refer to the RAD specification [20] for more information on these interfaces.

The RAD server prototype extends *PolicyEvaluatorLocatorBasicAdmin* into the *PolicyEvaluatorLocatorAdminExt* interface. This interface provides mechanisms to shutdown the PEL object. However, unlike the *AccessDecision* interface, *PolicyEvaluatorLocator* is not extended. This is because *PolicyEvaluatorLocator* itself provides mechanisms to get references to its administrative interface [20].

The object that implements *PolicyEvaluatorLocatorAdminExt* and *PolicyEvaluatorLocator* is *PolicyEvaluatorLocatorContext* (in edu.fiu.cadse.rad.pel package), and its implementation follows tie approach (Figure 11). At the time this report is written (2000), *PolicyEvaluatorLocatorContext* returns

default DC and PE objects. However, *PolicyEvaluatorLocatorContext* can become more complex and return different DC and PE objects depending on the resource name.

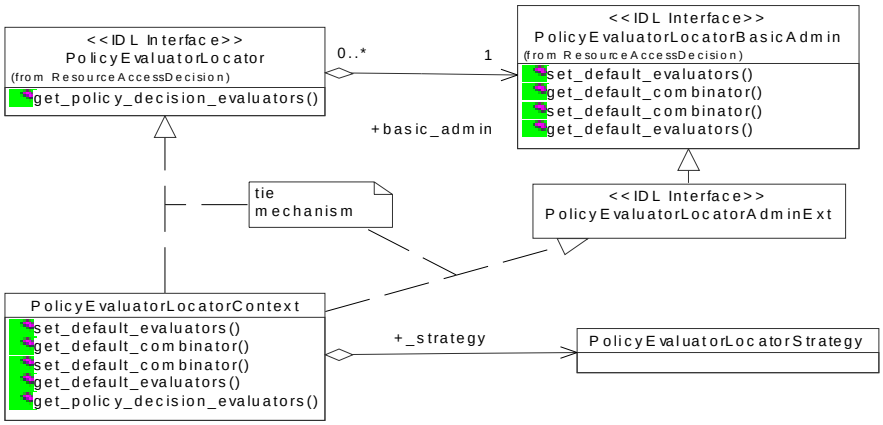

Figure 11: Implementation of PolicyEvaluatorLocator

Resource Name Patterns

A resource name pattern describes a name pattern composed from combinations of symbols using regular expression syntax as defined in [1]. For example, the regular expression (resource name pattern) *"a*"* would match all strings (resource names) that begin with the character *'a'*. For more information on resource name patterns, please see [20], and for regular expression syntax and usage, please see [1][10].

4.4. Dynamic Attribute Service

DynamicAttributeService (DAS) is represented by the interface of the same name. Unlike the previous components, the DAS does not have an administrative interface [20]. However, the RAD server prototype introduces a DAS administrative interface, *DynamicAttributeServiceAdminExt*. As in the design of PEL, tie approach is used to design a *DynamicAttributeServiceContext* object that implements the extended runtime and administrative interfaces of the DAS (Figure 12). The design of *DynamicAttributeServiceContext* follows a *Strategy* pattern; the mechanisms used to obtain the principal's security attributes applicable for an authorization decision are implemented by an object of type *DynamicAttributeServiceStrategy.* This is because the nature of security attributes and the means to manipulate and retrieve them can change from one organization to another.

When the ADO computes an authorization decision, it contacts the DAS (using the DAS get_dynamic_attributes operation), and passes to the DAS a list of security attributes, a resource name and an operation name (step 4, Algorithm 1). These three values are then passed to the object implementing the strategy interface (_strategy relation, Figure 12) which eventually returns a list of security attributes applicable to the authorization decision to the ADO. An implementation of DAS is free to add, remove or replace security attributes from the original list [20]. Furthermore, the security attribute list returned by DAS can change from one authorization decision to another. This dynamic nature of DAS influences implementations of the DynamicAttributeServiceStrategy interface.

RAD prototype includes one such implementation, the *EchoingDynamicAttributeService* object (Figure 12). The function of *EchoingDynamicAttributeService* is to return the same security attribute list it receives from the ADO. This implementation can be used to model the situation when there are no dynamic attributes (or when they are not needed) to obtain a authorization decision. Other implementations of *DynamicAttributeServiceStrategy* are possible which may follow a design using *Template* pattern (Figure 7).

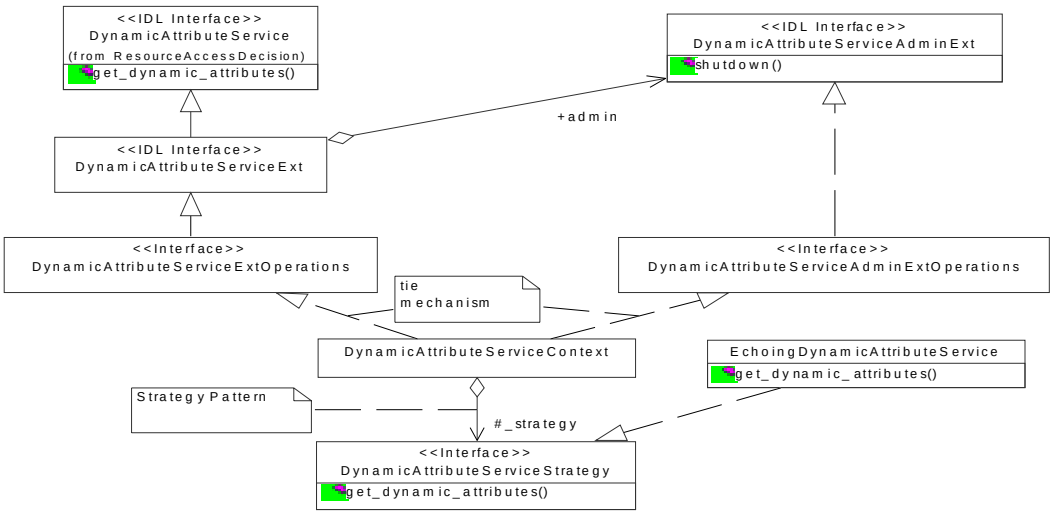

Figure 12: Dynamic Attribute Service

4.5. Decision Combinator

Compared to the other RAD components, *DecisionCombinator* (DC) has the simplest design. In the RAD prototype, DC does not have an administrative interface, nor does it have an administrative extension interface as the DAS component. A *DecisionCombinatorContext* class implements *DecisionCombinator* IDL interface using *Tie* approach (Figure 13). Although it would be simpler to use inheritance for the implementation of the *DecisionCombinatorContext* class, its tie-based design would ease the addition of an administrative interface if there is a need for it.

DC encapsulates the policy that dictates how to combine decisions results from multiple *PolicyEvaluators* (PE) into a single authorization decision [12][20]. This "decision combination" logic is delegated to an object implementing *DecisionCombinatorStrategy* interface. The RAD server prototype includes an object implementing this interface, *AbstractAndOrCombinator.* This class implements a logical AND on decision results obtained from multiple PE components. *AbstractAndOrCombinator* is further refined (using *Template* pattern) with *OpenWorldAndOrCombinationPolicy* and *ClosedWorldCombinationPolicy* classes (Figure 13).

In RAD prototype, *OpenWorldAndOrCombinationPolicy* returns "YES" (access granted) if all PE objects return "YES" or "DON'T KNOW". On the other hand, *ClosedWorldAndOrCombinationPolicy* implements a stricter policy: grant access only if all PE objects return "YES". *AbstractAndOrCombinationPolicy* is only one example of how *DecisionCombinatorStrategy* can be implemented. Other examples of implementations are strategy classes using majority votes or hierarchies of PE's where the decision result of one PE can override the decision results of other PE's.

4.6. Policy Evaluator

The function of a PE is to evaluate one or more encapsulated access policies to obtain an access decision on a resource given a list of principal's security attributes and an operation name [12][20]. In the prototype, *PolicyEvaluator* (PE) has the most complex design of all. As with most RAD components, it has run-time and administrative interfaces with respective extensions. Based on *Tie* Approach, *PolicyEvaluatorContext* implements both run-time and administrative interfaces. The design of PolicyEvaluatorContext uses *Strategy* pattern as it relies on *PolicyEvaluatorStrategy* interface for evaluation of policies (Figure 14).

RAD prototype has an implementation of *PolicyEvaluatorStrategy* interface, *AlwaysGrantDenyAbstractEvaluator* which serves as a template for *AlwaysDenyEvaluator* and *AlwaysGrantEvaluator.* As their names imply, *AlwaysDenyEvaluator* always denies access to any resource whereas *AlwaysGrantEvaluator* always grants access.[1] More complex implementations of *PolicyEvaluatorStrategy* are possible; examples of these are filesystem permissions and role-based access control (RBAC)[2] evaluators (see Figure 14). How such evaluators can be implemented is outside the scope of this report, and are left as part of future refinements of RAD prototype.

To know what access policies to evaluate given input parameters (from DC), a *PolicyEvaluatorContext* must maintain relationships between access policies and resource names. The implementation of these associations is based on *Strategy* pattern.That is, *PolicyEvaluatorContext* delegates the implementation of such associations to a class implementing the *PoliciesByResourceNameMap* interface (see Figure 14). By using this interface, developers can implement associations using any form of storage suitable to their needs independently of the implementation of *PolicyEvaluatorStrategy.*

RAD prototype provides a default implementation of *PoliciesByResourceNameMap*, *NullPoliciesByResourceNameMap* (see Figure 14). This implementation follows what is known as *Null Object Pattern* [13]. *NullPoliciesByResourceNameMap* implements a do-nothing version of *PolicyByResourceNameMap* interface. Such object relieves *PolicyEvaluatorContext* from testing for null values before accessing the methods of *PolicyByResourceNameMap* [13].

5. SUMMARY

CORBA Security service provides general-purpose infrastructure with which to build secured object-oriented distributed systems. However, complex application domain specific authorization decision logic are difficult, if not impossible to de-couple from application logic using only CORBA Security service. To overcome this limitation, at the time this report was written (2000), OMG adopted a Resource Access Decision (RAD) facility as a pre-final standard. By using RAD facility, developers can implement systems with authorization logic decoupled from application-specific logic and decentralized evaluation and administration of the access policies. The RAD facility complements CORBA Security access model, and allow developers to implement access control mechanisms of arbitrary granularity.

A prototype of the RAD server has been implemented to study the validity of the framework, to conduct experiments in the research of distributed access control and reason about properties of application systems using the prototype and of the prototype itself. Also, the RAD prototype provides simple, default algorithms for policy evaluation, decision combination, and acquisition of dynamic attributes. Since the design of the prototype is heavily influenced by design patterns, the prototype can easily be maintained and augmented with more complex access control mechanisms.

1. Since the decision result of a PE is a ternary value [20], another possible implementation would be a policy evaluator strategy that always returns “don’t know”.

2. Access control disciplines are explained in [22]. For more information on role-based access controls in particular, please refer to [23].

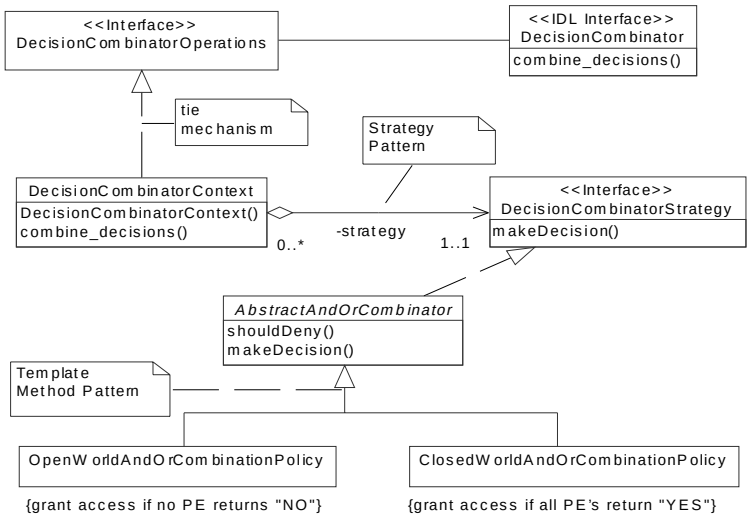

Figure 13: DecisionCombinator

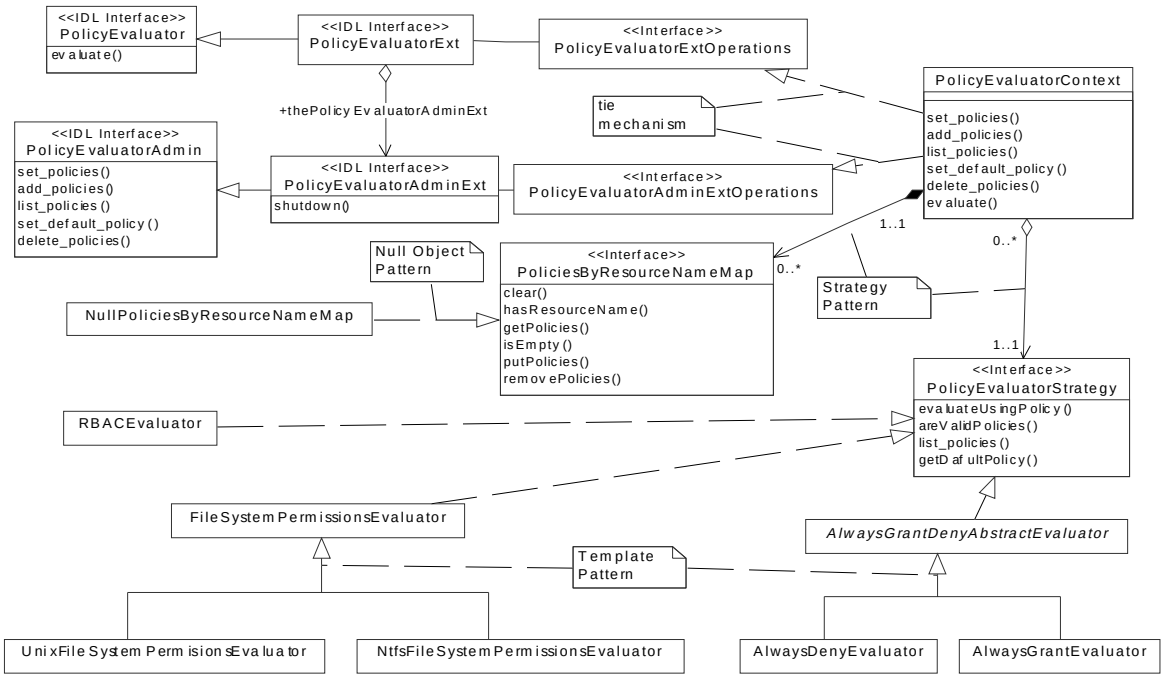

Figure 14: PolicyEvaluator

6. REFERENCES

[1] 9945-2:1993 (ISO/IEC) Information Technology-Portable Operating System Interface (POSIX)-Part 2: Shell and Utilities IEEE/ANSI Std 1003.2-1992 & IEEE/ANSI 1003.2a-1992 Section 2.8, pg. 77-91, “Regular Expression Notation.” Quoted in [20].

[2] Alexander Ran. Architectural Structures and Views. In Proceedings of the 3rd International Workshop on Software Architecture. Orlando, November 1 - 5, 1998, pg. 117 - 120.

[3] Butler W. Lampson. Protection. In Proceedings of the 5th Princeton Conference on Information Sciences and Systems. Princeton, 1971, pg. 437.

[4] David Garlan and Mary Shaw. An Introduction to Software Architecture. CMU Software Engineering Institute Technical Report CMU/SEI-94-TR-21, ESC-TR-94-21. In Proceedings of Advances in Software Engineering and Knowledge Engineering, Volume I. World Scientific Publishing Company, New Jersey, 1993.
http://www.cs.cmu.edu/afs/cs/project/vit/ftp/pdf/intro_softarch.pdf

[5] Doug Pedrick, Jonathan Weedon, Jon Goldberg, Erick Bleifield. Programming with VisiBroker: A Developer’s Guide to Visibroker™ for Java®. Wiley Computer Publishing. New York, NY, 1998

[6] Edward Amoroso. Fundamentals of Computer Security Technology. Prentice Hall. Upper Saddle River, NJ, 1994.

[7] Erich Gamma, Richard Helm, John Vlissides and Ralph E. Johnson. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In Proceedings ECOOP’93, O. Nierstrasz (Ed.), LNCS 707, Springer-Verlag, Kaiserslautern, Germany, July 1993, pg. 406-431

[8] Erich Gamma, Richard Helm, John Vlissides and Ralph E. Johnson. Design Patterns: Elements of Reusable Object-Oriented Design. Addison-Wesley, Reading, MA, 1994.

[9] Jeff Kramer. Distributed Software Engineering: A State of the Art. In Proceedings of The 16th International Conference on Software Engineering pg. 253-263. Sorrento, Italy, 1994.

[10] Jeffrey E.F. Friedl. Mastering Regular Expressions. O’Reilly, Cambridge, MA, 1997.

[11] Jim Waldo, Geoff Wyant, Ann Wollrath, Sam Kendall. A Note on Distributed Computing. Technical Report TR-94-29. Sun Microsystems, November 1994. http://www.sun.com/research/techrep/1994/abstract-29.html

[12] Konstantin Beznosov, Yi Deng, Bob Blakley, Carol Burt and John Barkley. A Resource Access Decision Service for CORBA-based Distributed Systems. In Proceedings of the Annual Computer Security Applications Conference, Phoenix, Arizona, U.S.A, December 6-10, 1999. http://www.cs.fiu.edu/~beznosov/doc/acsac-1999-paper.pdf.

[13] Mark Grand. Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML. Vol 1. Wiley Computer Publishing, New York, NY, 1998.

[14] Len Bass, Paul Clements and Rick Kazman. Software Architecture in Practice. Addison-Wesley, Reading, MA, 1998.

[15] Jeff Magee, Andrew Tseng, and Jeff Kramer. Composing Distributed Objects in CORBA. In Proceedings of The 3rd Interna-tional Symposium on Autonomous Decentralized Sys-tems. Berlin, April 1997.

[16] Object Management Group. IDL to Java Language Mapping. June, 1999. OMG document number: formal/99-07-53.

[17] Object Management Group. CORBA 2.3 Specification. July 1998. OMG document number: formal/98-12-01.

[18] Object Management Group. CORBAservices: Common Object Services. July 1998. OMG document number: formal/98-12-09.

[19] Object Management Group. Healthcare Resource Access Control Request for Proposal. August 1998. OMG document number: corbamed/98-02-23.

[20] Object Management Group. Resource Access Decision (RAD). May 1999. OMG document number: cor-bamed/99-05-04.10.

[21] Ravi Sandhu and Pierangela Samarati. Access Control: Principles and Practice. IEEE Communications, 32(9), September 1994.
[22] Ravi Sandhu, Edward Coyne, Hal Feinstein, Charles Youman. Role-Based Access Control Models. IEEE Computer, Volume 29, Number 2. February 1996, pg. 38-47.
[23] Ravi Sandhu. Access Control: The Neglected Frontier (Invited Paper). In Proceedings of the First Australasian Conference on Information Security and Privacy. Wolongong, Australia, June 23-26, 1996.
[24] Urlich Lang. Security Aspects of the Common Object Request Broker Architecture. Master's thesis, Royal Holloway University of London, 1997. http://www.cl.cam.ac.uk/~ul201/mscdissertation.pdf