2 Requirements for Manageable Access Control

Chapter 1 has motivated the general need for a flexible, scalable and manageable access control infrastructure for CORBA, which rests on the assumption that application–level access control in distributed applications is not static and self–contained, but requires careful management. The basic assumption of this thesis is that handling application–level security policies is difficult and error–prone, and that it requires support that minimizes the potential for error.

This chapter examines the requirements arising from this assumption more closely and establishes some of the terminology used throughout this thesis. Sections 2.1 to 2.3 provide an overview of the tasks that deal with access policies at different stages of the application life cycle and discuss manageability issues and requirements. A general set of requirements is summarized in section 2.4.

2.1 Policy–related Management Tasks

To identify the problems faced by managers and the kind of support these problems require, it is necessary to examine the different roles in which managers operate directly or indirectly on access policies and the stages in the application life cycle where access policies must be considered. It is useful to take a deductive approach and begin by examining the later stages first to also identify the requirements on earlier stages.

Access control is performed when processes try to access protected objects at runtime. It will be shown, however, that policy issues must be addressed in earlier stages in the application life cycle to enable appropriate runtime management. This section describes the management tasks related to access policies at runtime. Subsequent sections examine the earlier stages of deployment and application development.

In general, an application security manager is responsible for monitoring applications for security breaches and for updating policy information in response to detected breaches or in response to changes in the security requirements or the application environments. Examples for such changes are users joining or leaving a company, reassignment of responsibilities within an organization, or the creation of new objects that require protection.

The three main tasks that can be identified correspond directly to what can be called the three basic dimensions of access control: principals, target objects, and authorizations. The

term *authorization* is used here as a synonym for privileges or access rights and it will later be extended to cover not only permissions, but also denials. This use of the term should not be confused with its other use as "checking authorizations" or "making an access decision", which is also frequently found in the literature. Informally, access policies are simply sets of related authorizations. The respective management tasks for each dimension — principal management, object domain management, and policy management — and their relationship to access policies are examined in the following sections.

2.1.1 Principals and Credentials Management

> *[An access decision function] needs a trustworthy way to know both the source of the request and the access rule. Obtaining the source of the request is called 'authentication'; interpreting the access rule is called 'authorization'. Thus authentication answers the question "Who said this?", and authorization answers the question "Who is trusted to access this?".* [Lampson et al., 1992]

The question that this section addresses is with which kinds of request sources a manageable access control infrastructure should be concerned. Whose accesses are the subject of access control decisions and who needs to be authenticated? It will be shown that answering these questions implies making explicit design decisions in the definition of access models. In [Lampson et al., 1992], the term used to describe request sources is *principal*. This is also the term adopted here.

Principals

Technically, requests in a distributed system are always initiated by processes and arrive at a target over potentially insecure *channels*. Processes are commonly called *subjects* in the security literature and this term is sometimes also applied to human users, but it is used here for active entities *within* a computer system — processes — exclusively. Human users interact with subjects in computer systems through external interfaces, and their actions are much less amenable to control than those of processes.

From the perspective of an access control mechanism at the target object side, the only immediately identifiable entity is the communication endpoint and thus the channel over which a request arrived. However, channels are a low–level, technical concept that is not suitable as a basis for access decisions. Consequently, other access control information needs to be obtained and authenticated, which then permits to deal with a more abstract notion of principals. Examples of principals to which it can be useful to refer in policies are "the NFS Server 'Big Blue' ", "temperature sensor 11", "the Accounting Manager", or "Working Group 7", i.e., a process, a device, a role, or a group. To see why role principals are useful, consider the set of authorizations that is required for a manager to approve orders in an order processing system. This set is not dependent on the personal identity of the individual manager, but rather on his job function. In cases like these, it is more convenient to assign authorizations to principals

that can model job functions, such as a role. Otherwise, a new manager has to be assigned all these authorizations individually, rather than just being assigned to the manager role.

While roles are a useful access control concept, there is no fundamental reason why principals should a priori be restricted to roles. The only requirement from the point of view of an access control mechanism is that it must be possible to attribute requests to them. To verify that a request originates from a particular principal, the receiver must *authenticate* the principal who claims to have spoken on a channel. In the terminology of [Lampson et al., 1992], a channel must present *credentials*, which are a proof that the channel "speaks for" a principal at this time. If a principal C speaks for another principal A, written as $C \Rightarrow A$, then every statement that C makes is also made by A. Technically, this can be achieved using public key cryptography and appropriate protocols, e.g., by using SSL transport connections on the channel, which allows the initiator of a request to establish a public key pair and a chain of X.509 public key certificates in an SSL session context. Without knowing anything about the holder of the public key announced in the context, the receiver can verify that the certificates apply to the established public key, that certificates bear the signature of a trusted[1] *certification authority* (CA) and that the corresponding private key is known at the other end of the channel. If this verification succeeds, this information can be trusted to be bound to the context, which proves that the channel speaks for the principal identified by this information. Note that this verification does not prove that the principal initiated the request or even that it is actually the source of the request. All it proves is that someone provided the channel with credentials and a private key.

For example, a certificate signed by a trusted authority on roles may bind the role name "Manager" to the public key in the context. If the channel speaks for this role "Manager," it can be concluded that, when the channel makes a statement, the role also makes it. Using more elaborate higher–level protocols to establish contexts, it is possible to verify that a channel speaks for a number of principals at the same time, if the sender has provided a set of different credentials. For example, the channel could speak for a role principal "Manager" and a "secret" clearance level principal.

It is important to stress that principals are an entirely abstract concept that has only been defined using the "speaking" metaphor for requests. Neither the public key infrastructure that is used to create certificates nor the access control mechanisms rely on any inherent semantic properties of principals, i.e., there is no technical difference between access requests from the principals "Gerald" or "fax machine 7" as long as processes are able to provide verifiers with the corresponding credentials. Technically, access policies are free to use whatever interpretation suits them best in any given context.

However, management requirements also need to be considered, and these restrict the choice of possible interpretations. A first management requirement for principals is that they need to be named in a way that makes "sense to people, since users can't understand alternatives like unique identifiers or keys" [Lampson et al., 1992]. As discussed in [Ellison et al., 1999a],

[1] We avoid any discussion of the notion of trust. For our purposes, trust is a manner of configuring verifiers to accept a particular signature.

requiring a manageable, globally unique name space of meaningful identifiers is not realistic, so there must be an infrastructure of local naming authorities that are trusted to assign meaningful names to principals. Identifying principals through names is thus performed relative to a given context of naming authorities.

Credentials Management

The abstraction of principals is realized by supplying and verifying credentials and these credentials must be created and managed. Typically, credentials in distributed systems are public key certificates [Ellison et al., 1999b], [CCITT, 1988b] and rely on *public key infrastructures* (PKIs). The task of managing credentials includes setting up certification authorities for the different types of credentials and controlling the creation and storage of credentials. Credentials management is not a subtask of applications management because credentials are usable across different application contexts. This section argues that large numbers of credentials for principals cannot be adequately managed without a well–defined and intuitive *intensional* definition of principals. An extensional definition of principals is sufficient for the description of access control mechanisms [Gollmann, 2000], but not for management.

When examining different possible interpretations of principals, it is important to consider the dual nature of principals in the worlds of credentials and of policies. In the general case, credentials are managed by user managers or PKI administrators, but policies are managed by application policy managers. While there is a clear separation of concerns between the two management roles, principals are the link between them. Since both management roles ultimately model real world aspects in terms of computer artifacts, there must be agreement on this notion between the two administrative roles. Otherwise, subtle mismatches between the way principal credentials are managed and the way they are used in policies might arise. The practical problems experienced in setting up cross–platform access control in large companies show that this is a real problem (cf. [Roeckle et al., 2000]).

In a typical organization, user managers are concerned with providing human users with the credentials that permit them to accomplish their tasks. Other active, abstract entities that need credentials of their own, i.e., long–running processes, are treated in a similar way to human users for management purposes. For large populations of users, managers obviously need aggregational concepts that allow them to quickly navigate the set of users and credentials, and also support management actions at larger granularities, such as assigning or removing some set of credentials collectively to a set of users.

A typical way of assigning credentials in current distributed systems is to store them in repositories such as provided by the X.500 [CCITT, 1988a] or LDAP [Yeong et al., 1995] directories. These support hierarchical modeling of organizations and thus allow managers to navigate the user population using structures with which they are familiar, i.e., organizational structures like branches, departments, and groups. For user managers to be able to manage credentials for principals, principals should either directly correspond to these concepts or integrate well with them. As an abstraction, principals must insulate user managers from all

details of individual applications and their policies, but their credentials must still be manageable in a way that is consistent with access policies.

Some types of principals may define interesting relationships, e.g., a seniority relation between roles or subgroup relations between groups. Role–based access policies may rely on role relations for implicit authorizations that a role inherits from its junior roles, so credentials managers must understand the semantics of principals and their relationships in order to manage credentials properly, otherwise policies relying on these semantics and relationships cannot be enforced. Consequently, credentials must either be intuitive to manage or managers need concise documentation about how credentials are used for access control purposes.

2.1.2 Object and Domain Management

For an access decision function to be able to answer the question "is this access to this target allowed?" it must be able to retrieve the authorization information that applies to the current access and target. Managing potentially large numbers of objects and their associations to security policies requires support for aggregating objects that are treated alike for management purposes. The common management concept used for grouping managed objects is a *domain* [Sloman and Twidle, 1994], which allow managers to structure large systems into manageable subsystems. At the same time, domains can be used to model the diverse spheres of management authority and responsibility that are frequently found in large organizations.

Because domains are an established and flexible concept for managing policy assignments for objects, we state the requirement that object management should be domain–based for security purposes. It is an obvious but important requirement that a domain concept for CORBA objects and their policies integrates with the objects' life cycles. For example, it must be possible to protect objects immediately after creation, which means there must be means to associate an object with a policy before the object is accessed for the first time. It should also be possible to dynamically associate objects with new policies. Chapter 3 examines CORBA domain concepts in more detail.

For simple cases where only a single application–wide policy domain or just a single system–wide domain exist, it can be argued that managing objects and their associations with access policies is actually a subtask of policy management and not a management task in its own right. In cases where domain hierarchies are used to model refinement of policies from a global system policy via different levels of responsibility down to a local application domain, this management task goes beyond the authority of a single application policy manager.

It can also be argued that even in cases where complex domain structures are used, the relationship between objects and policy domains is unlikely to undergo dynamic changes after it was established and thus, that setting up domains is actually a deployment rather than a management task. Depending on how frequent changes in the domain structure and the assignment of objects to policy domains are in practice, this is a tenable position, especially if there is no API that would allow applications to reconfigure this structure so that monitoring the object–domains relationships is not necessary. In the general case, however, where the

complexity of the domain structure and the frequency of changes in the object–domain mapping are unconstrained, object and domain management can be regarded a management task in its own right. This task is obviously security–sensitive and requires managers to understand the consequences of, e.g., moving an object from one policy domain to another, or of creating additional domains and inserting these into a hierarchy of existing policy domains.

Domain management requires a security domain membership service [OMG, 1998b], [Brose et al., 2001a], that is responsible for creating and managing policy domains. It is also responsible for assigning and removing objects to and from domains and is concerned with managing the relationships between domains. Relationships between domains can be used to model areas of administrative responsibility. Finally, the association of domains with policies must be managed.

2.1.3 Policy Management

Policies were informally defined above as "sets of related authorizations," but the nature of this relationship between authorizations was not explained. The minimal policy consists of a single access rule and the maximal application policy could be the set of authorizations that describes all relevant accesses to all application objects. In fact, the aggregation of objects or authorizations into a single whole is a design decision and there are no clear rules for grouping authorizations other than that they are perceived as belonging together.

Policy management is the task of monitoring policies and potentially modifying them in response to detected problems or changing requirements. If the managed policy is purely application–specific, this is a subtask of application management. In multi–policy environments, however, this task is also performed at higher levels, where more general policies apply to objects of more than one application.

Authorization Changes

Policy managers may need to modify authorizations for a number of reasons:

1. A manager monitoring an application might detect that the original policy does not enforce the security requirements it was intended to meet, either because it is incorrect or incomplete. For example, a policy might not have authorized certain accesses which are required for some of the application's functionality or it might have authorized too many accesses that are actually considered unsafe. Consider an object representing orders where principals should be able to cancel the order later, but are not authorized to call the cancel() operation to do so. On the other hand, principals might have inadvertently been authorized to call cancel() on all order objects in the system, not just a particular subset. In both cases, managers need to adjust these policies.

2. Policies may need to change in response to changes in the environment, i.e., when new objects or principals are introduced into the system. For example, it might have been

deemed necessary to introduce new roles for user management reasons, which were not anticipated by the original policy description. A policy manager will then have to select the appropriate authorizations and assign them to the new role.

3. Policies can change because of changes in the security requirements. On the one hand, an organization might have decided to put more trust in the members of certain existing roles and therefore to authorize these roles for more accesses. On the other hand, stricter security requirements might demand that certain interactions may only be carried out by more senior roles, so that authorizations have to be removed from one role and assigned to another.

4. Rights changes might be required by the applications themselves, e.g., when an object in a workflow application passes from one processing stage to another so that authorizations applying in earlier stages need to be removed, whereas new ones are required in the current stage. For example, in an order processing system, customers could be authorized to cancel an order any time until receipt of the order is acknowledged by a supplier, at which time the permission to call cancel() is removed. This class of changes is actually not appropriate for external management because the changes happen regularly during execution of the application and are closely tied to the application's logic. It could be argued that policies of this kind should not be subject to management at all but rather hard–coded in the application itself. However, such an approach is undesirable because the application's behavior in terms of allowed and denied accesses is no longer represented by an external policy and, moreover, can no longer be monitored or modified by policy managers in case of problems or interference with additional external policies.

The management activities carried out in response to these changes are basically the same as those performed by the deployer role, which is described in section 2.2, and will rely on the same set of tools. It is therefore possible that the deployment and management roles might be assigned to the same parties. Changes during system operation will be evolutionary rather than revolutionary, however, so management is expected to require less coordination and cooperation between managers than deployment.

Policy Representation

Even disregarding this latter case, it is apparent that policy managers must understand the individual authorizations in a policy. Otherwise, they won't be able to react properly to potential problems or perhaps not even recognize policy violations in the application's protection state. Without also understanding the intended semantics of the policy as a whole, it is unlikely that management actions will be able to preserve the original policy intentions. The requirement that arises from these observations is that policy managers should interact with policies in a declarative representation which captures policy semantics at an abstract, mechanism–independent level [Woo and Lam, 1993] and supports management in terms of concepts that are familiar to application managers.

Because the traditional access matrix is very sparse, its authorizations are usually stored either row–wise or columnwise rather than as a whole. Matrix rows represent a principal's authorizations for objects and can be stored in the form of *capability lists* in the subjects's address space, whereas matrix columns represent all access rights for an individual object and can thus be stored together with the object as an *access control list*. Both representations have their advantages in different situations, but they do not lend themselves to a centralized abstract representation that would make a policy comprehensible as a whole because the underlying Lampson matrix model does not provide concepts that would support the aggregation of simple authorizations into higher–level authorizations, the definition of structural relationships between authorizations, or the definition of constraints.

From a more general point of view, any access control model implicitly defines an *authorization language* that can be used to specify access control policies.[2] If the concepts and abstractions of this model are not well designed, this has direct bearings on the quality of the policies that can be written using that model and on the relative ease or difficulty of managing these policies. The design of access control models has traditionally been dominated by the two competing design goals: *assurance*, i.e., the formal tractability of security properties, and *expressive power*. The general observations outlined in the preceding sections, however, call for an access control model designed for manageability purposes and according to programming language design principles. Such a model should support both convenient textual or graphical syntax and appropriate management abstractions. To be practically useful, any such model must also represent a reasonable compromise between expressiveness and verifiability, i.e., a model that is manageable but cannot express practically interesting policies or cannot give any formal guarantees cannot be considered an enhancement over existing approaches.

Application managers — including application security managers — are expected to have a general understanding of an application's function and of the common scenarios in which the application functionality is used. They cannot be expected to understand the entire internal application design, let alone the complex interactions between objects in a distributed, object–oriented application. It is thus essential to define abstractions that let managers concentrate on accesses in a way that corresponds to the actual use of the application and its resources and preferably not care about individual operations in object interfaces. Note that this requirement does not imply granularity restrictions on access *mechanisms*, nor even precludes management of fine–grained authorizations for individual operation accesses. It should thus be possible to specify use case oriented aggregations of authorizations in the language defined by the access model. Moreover, it should be possible to define compositions and refinements of these aggregations. Both concepts are well–known in programming language design and support reuse of definitions and structured development.

Policy managers have to monitor and potentially modify policies, but they should not have to define them. In particular, they should not have to design the aggregate authorizations mentioned in the previous paragraph because this, again, requires understanding the application

[2] More precisely, the access model does not implicitly define one single language because it is possible to define any number of languages that each come with their own syntax. Semantically, these languages are all equivalent.

design down to the level of individual operations. We therefore state that an application policy manager receives a valid initial policy configuration after the application is deployed.

2.2 Policy Deployment

The usual notion of deployment includes tasks such as the initial installation and configuration of a system, but it also includes software updates and removal operations [Carzaniga et al., 1998]. With respect to access control, this discussion concentrates on the initial tasks and considers update and removal as management tasks. As required in section 2.1.3, one of the results of the deployment stage is an initial policy configuration.

Deployers, like managers, cannot be expected to understand the internal details of an application's design. Consequently, they must rely on an abstract policy specification or a policy template to be delivered with the application. During application deployment, abstract specification concepts are refined and potentially modified to suit the actual operation environment. Obviously, this only works seamlessly if the access models used to describe the policy design and the model underlying the actual protection mechanisms in the target environment are compatible. Note that the "hand-over" interface between policy developers and deployers may disappear if deployers are application experts charged with deployment. This is commonly the case if deployment is performed by the application vendor's organization or a support contractor, rather than internally by the organization that eventually operates and manages the application. In this case, the two roles are effectively merged. Since the final result of the deployment stage is a valid, initial policy configuration that provides high–level management abstractions, the conceptual gap between these abstractions and actual runtime entities must still be bridged and the language–based approach described here still appears to offer the best support for the overall process.

The only party that is competent to define the required policy abstractions are the application developers who designed the object interfaces and understand the semantics of individual operations. Consequently, the required access language must allow developers to provide a static policy specification that is independent of the target environment and is delivered with the application. This specification may be refined during deployment in the target environment and managed during the lifetime of the application.

Policy deployment involves mapping the principals that are referenced in the static policy to the actual principals in the target environment and configuring the access decision functions to rely on the appropriate certification authorities. It might further involve creating policy domains for application objects and associating these with policy representations. If the application is deployed in a multi–policy environment, deployers have to find out about applicable system policies and potentially resolve conflicts between existing and new policies. All of these tasks require precise policy documentation and appropriate tool support.

2.3 Policy Issues in Application Development

As stated in the previous sections, the deployer of an application–specific policy needs an access policy description. It was argued that this document should describe accesses at a more abstract level than individual operations on individual objects because this would make deployment too complex. This additional document is called *policy design* and has to accompany the final software product, e.g., in the form of a descriptor file that can be read by deployment and management tools. It is thus an important product of the software development process.

This section examines those areas in software development that are critical for the final access policy design document. The software development process is assumed to be generally based on refinement in some kind of waterfall model, but this assumption is not critical for the discussion. This section does not reveal any new requirements for a manageable access control infrastructure as a whole. Rather, it characterizes some of the concepts of an appropriate policy description language and suggests how security policy development can be integrated into the more general software development process.

2.3.1 Requirements Analysis

> *Requirements engineering is the most critical task of managing secure system development, and is also the hardest.* [Anderson, 2001, p. 503]

The first stage in any of the common models of the software development process [Scacchi, 2001] is the analysis of the requirements a software system has to fulfill. Generally, the requirements identified in this stage can be divided into *functional* and *non–functional* requirements. Functional requirements describe what the software system is supposed to do in terms of its functionality, whereas non–functional or quality requirements specify how and under which conditions this functionality is to be displayed. Typical non–functional requirements are concerned with performance, accuracy, user–friendliness, or security. Non–functional requirements are generally hard to analyse and model because they are “hard to express in a measurable way”, “tend to be properties of a system as a whole” [Nuseibeh and Easterbrook, 2000], and are hard to integrate with the standard requirements process [Devanbu and Stubblebine, 2000], [Baskerville, 1993]. The typical approach to developing security policies is the following:

> The engineering of a security policy starts with a risk analysis and ends with an executable code unit that is ready for integration into a security architecture. Risk analysis identifies threats and security weaknesses of an IT–System or application and results in a set of security requirements. [Halfmann and Kühnhauser, 1999]

This approach is illustrated in figure 2.1 from [Halfmann and Kühnhauser, 1999] and relies on specialized security models for the specification and implementation of policies. Comparatively little work, however, has been done on integrating security requirements into general models of software engineering [Chung, 1993].

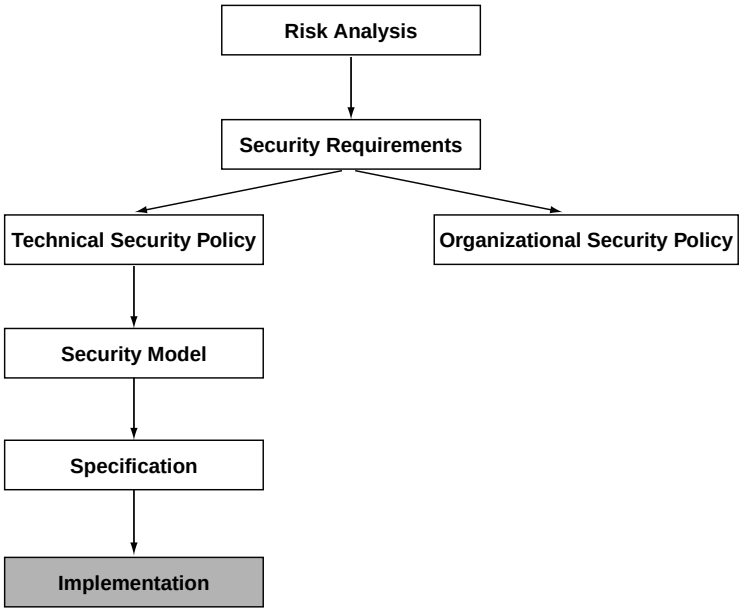

Figure 2.1: Steps in the Development of a Security Policy.

> Clearly, there is much to be gained by developing processes and tools to unify security policy development into the system development process, specifically by making use of system models when designing security. One attractive approach is to adopt and extend standards such as UML to include modeling of security related features such as privacy, integrity, access control, etc. [Devanbu and Stubblebine, 2000]

An important factor in this context is developer acceptance for security because "for software developers, security interferes with features and with time to market." [Lampson, 2000] Making use of one of the modeling languages of the *Unified Modeling Language* (UML) [OMG, 2000b] to support the definition of access control requirements is an approach for integration of security that is proposed in the following. The only other attempt of integrating UML with security of which the author is aware is [Jürjens, 2001], which discusses annotating class diagrams with sensitivity labels for multi–level security.

For a number of reasons, access control appears less difficult to integrate with standard software development models and techniques than other security aspects. First, access control requirements can generally be expressed in a "measurable way" because an access is either granted or denied. Even though the general safety property has been shown to be undecidable [Harrison et al., 1976], it is simple to say "how well" an individual access operation should be protected. Second, protection is not a property "of a system as a whole" because accesses refer to distinct system components. If the units of access control can be identified with the units of system design, protecting access to the system can be broken down to protecting its indi-

vidual components. Third, since access control is only concerned with the external interfaces of protected resources it blends well with data abstraction techniques and standard object–oriented models. The following paragraphs examine how use cases can be used to identify and document access control requirements for object–oriented applications.

Use Cases

Use Cases are a simple behavioral abstraction that is used in object–oriented analysis to describe system functionality from an external perspective [Booch et al., 1998]. Use cases are informally defined in the specification of the UML [OMG, 2000b] as

> [...] representing a coherent unit of functionality provided by a system [...] as manifested by sequences of messages exchanged among the system and one or more outside interactors (called actors) together with actions performed by the system. (p. 3–89)
>
> [...] An actor defines a coherent set of roles that users of an entity can play when interacting with the entity. An actor may be considered to play a separate role with regard to each use case with which it communicates. (p. 3–90)

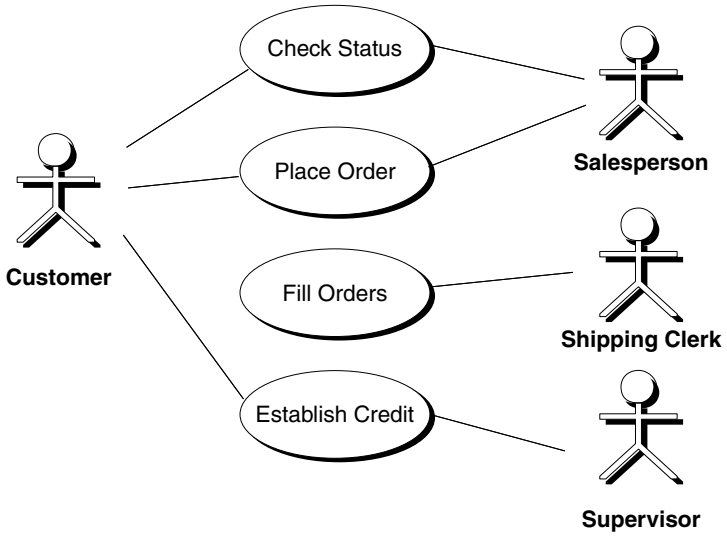

Figure 2.2: A Use Case diagram.

Use cases are used to abstract from the internals of a system (an “entity”) and focus on its behavior at its interfaces with “outside interactors,” which “represent both human users and other systems” (p. 2–125). Figure 2.2 shows a simple use case diagram from [OMG, 2000b] that illustrates four use cases, which are depicted as ellipses, and four actors. In a complete use case analysis, the diagram would be complemented by a textual description of a sequence of actions that comprise the use case, e.g.,

1. lookup item number

2. check item availability

for the “Check Status” use case in figure 2.2. Action sequences may contain optional steps and branches. A concrete instance of a use case which performs a particular sequence of actions is called a *scenario*, so a use case can also be viewed as a class of scenarios.

Use cases are an informal technique for requirements analysis. There is no formal process for the refinement of use cases into class and activity diagrams and UML defines no conformance points that could be used to generate proof obligations. Use case elements are only given broadly intuitive rather than formal definitions. Because of this apparent intuitiveness — and despite this imprecision — use cases are a useful vehicle for identifying and describing system functionality during object–oriented analysis, where they serve as a means of communication between analysts and customers.

The main point here is that use case descriptions are focusing on those system aspects that are the most important for access control, viz. which principals (actors) are accessing which targets (entities). A use case analysis of the functionality of a projected software system can serve as an important first step for the specification of protection requirements. While such a document does not yet take any environment–specific security requirements — such as those arising from risk analysis — into account, it represents a high–level access policy based on the *need–to–know* principle. This view implies that *every* entity in the system is regarded as a protected resource and that every access that is not part of a defined use case has to be denied.

If this approach is not already sufficient to counter the perceived risks, it can be complemented by what could be termed “misuse cases”, which can serve as documentations of potential attacks that the system must be designed to withstand, i.e., unauthorized accesses that must not be allowed. Explicitly documenting accesses that must not be authorized is useful even if there are currently no use cases allowing these accesses. Such a set of denials can serve as a constraint that ensures that an evolving policy remains “safe”, i.e., that management errors such as wrongly assigning authorizations do not result in policy violations: management tools could detect the conflict between the positive and negative authorization and report an error.

The main advantage of a use case analysis from the perspective of access control is the identification of actors, which can also be interpreted as principals, and the abstract identification of the system interface, which is later refined in terms of object interfaces. In addition, use cases could be delivered as an abstract documentation of system use to application managers, provided that the documents are kept consistent with subsequent modifications in the design and implementation stages. Use cases are not a panacea, of course, but simply a practical approach of identifying authorized and unauthorized accesses to application resources. The case study in chapter 6 will show concrete examples of how use cases support the identification of principals and the interfaces that need to be protected.

2.3.2 Design

> *The definition of the access rights should be regarded as an important step of the system design; it would constitute an important system design document.*[Linden, 1976]

Generally, application design takes the output of the analysis stage (use cases, analysis models, other requirements descriptions) as input and produces a design model as output. As suggested here, the design model should include a policy design that is suitable for application deployment. The policy design can be ideally regarded as complete at this stage because the concepts with which it is concerned are not modified during the subsequent implementation stage, unless the system design itself is modified because inadequacies are detected later. In this case, the policy design has to be modified to remain consistent with the system design.

For CORBA applications, the externally visible interfaces are defined in IDL during the design stage. Any non–trivial application will also have internal interfaces that are not defined in IDL, but it is assumed here that actors always interact with the system through IDL interfaces, so these IDL interfaces are the units of protection that have to be addressed in the policy design. Not all IDL interfaces are equally relevant at this stage for the design of an access control policy, however, because some of them might have been designed as internal interfaces to components that are not externally visible. Assuming a closed policy, i.e., one where all accesses that are not explicitly allowed are forbidden, it is sufficient to consider only those interfaces with which principals directly interact. These external interfaces can be regarded as a refinement of the abstract system interfaces in the use cases of the analysis stage. The correctness of this refinement step can generally not be formally proved since the original models have no formal definition, but it can be checked that these interfaces provide operations to support the “sequences of messages” specified in use cases. These can now be represented in more detail using, e.g., UML interaction diagrams. The set of actors was already complete after requirements analysis and is not refined during design (unless, of course, the analysis is found to be faulty or incomplete).

The central issue in this discussion is which concepts should be used for the expression of the policy design, i.e., what is a principal and what is a protected object. One potential approach is to map actors and IDL interfaces to corresponding concepts in a sufficiently expressive access model, e.g., [Jajodia et al., 1997]. This would not be an *integrated* development approach, however, because it would introduce a separate security model and a mapping from design entities to security concepts into the development process. It can be argued that this mapping could be performed automatically by dedicated tools and that a separate team of policy designers could specify the policy relying only on the requirements expressed in use cases. Thus, developers do not have to deal with two models. However, the same problem of having to deal with both an object model and a separate security model would surface again for application management. This approach thus does not appear to be practical, since application management must be assumed to be based on actual CORBA objects rather than on a specialized security model.

For these reasons, we state the requirement that the security policy should be expressed and later managed directly in terms of design artifacts, i.e., access operations in CORBA IDL interfaces and UML actors. Another benefit of this more integrated approach is that it is likely to be practically more acceptable for development teams. Moreover, static typing of IDL interfaces can more directly be leveraged for statically checking policy descriptions for consistency. Support of a type checker for policy specifications is an important requirement here.

The integrated approach requires an access control model that supports a seamless translation from actors to principals, so its notion of principals should be semantically close to the concept of actors, which were defined as a "coherent set of roles". It is important to note that UML defines actors exclusively through their interaction with the system, i.e., through their logical function in a particular context. Thus, actors abstract from individual users in the target environment, which cannot be known until deployment. In UML, actors may have generalization relationships to other actors. Generalization does not entail structural relations between actors, such as organizational position, however, but is a purely behavioral notion. Its semantics is that a more specialized actor can play the same roles as the more general actor.

The most straightforward approach to an interpretation of principals that supports an integration of security policy design with standard object–oriented modeling techniques is thus that of a role, where roles are defined as logical functions in interactions. In section 2.1.1, it was argued that a principal concept should either map to or intuitively integrate with organizational structure. As defined here, the actor meaning of role is not a structural notion, so it should not be directly mapped onto, e.g., groups. However, roles can be made intuitive to user and credentials managers as a "job function" or "task".

2.4 Summary of Requirements

This chapter argued that an access control infrastructure for large–scale systems must support separation of concerns between the different management roles of user credentials management, object domain management, and policy management. To do so, it must insulate managers from details that are irrelevant for their task. At the same time, it must define common concepts that enable communication between managers. One important concept that managers must agree on is the notion of principal, which must be intuitive and integrate with organizational user management concepts such as groups.

It was argued that it is not sufficient to only support runtime management since managers cannot be expected to define application–level policies. Consequently, managers must rely on documentation provided by deployers and application designers, who are charged with delivering an abstract policy design document together with the application. The most straightforward approach to support this process is to define a policy language that provides adequate concepts. Such a language should also be usable as the basis for runtime management and support an integrated development approach. The requirements for the design of this language are:

- It should support an abstract representation that makes the underlying semantics of the

policy as a whole comprehensible to managers. At the same time it should shield them from unnecessary, application–internal detail.

- It should be based on roles as principals to support a seamless integration of policy design with application design based on use cases with actors.
- Its authorization concepts should directly refer to operations and also support negative authorizations to document unauthorized accesses.
- It should support typing and type checks to catch specification errors.
- It should be sufficiently expressive to support a wide range of practical policies.

Finally, an important concern is that this language is both practically implementable and that runtime access checks relying on its concepts can be performed efficiently.