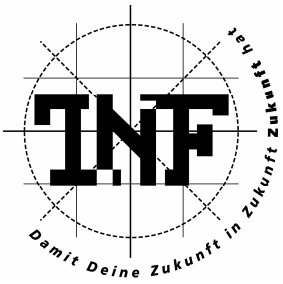
JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis



GAMMA – A platform independent framework for reusable authentication, authorization, and auditing components

Dissertation zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

im Doktoratsstudium der *technischen Wissenschaften*

Angefertigt am Institut für *Anwendungsorientierte Wissensverarbeitung*

Betreuung:

a.Univ.-Prof. Dr. Josef Küng

Von:

Dipl.Ing.(FH) Stefan Probst

Gutachter

a.Univ.-Prof. Dr. Josef Küng

a.Univ.-Prof. Mag. Dr. Werner Retschitzegger

Linz, *Oktober, 2004*

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

_____ _____

Datum, Ort Unterschrift

Kurzfassung

Sicherheit ist in der heutigen weltweit vernetzten Zeit zu einem wichtigen Bestandteil jeder Softwareapplikation geworden. Obwohl wir aus dem Bereich der Mainframe Computer über bewährte Konzepte für Zugriffskontrollen verfügen, fehlen uns immer noch ausreichende Mechanismen die in modernen Softwarearchitekturen eingesetzt werden können. Zudem können wir die Situation beobachten, dass Sicherheit oftmals nachträglich in existierende Produkte eingebaut wird, was oftmals zu Problemen in der Software oder in den Sicherheitsmechanismen selbst führt. Ein Grund dafür könnte das Fehlen ausreichender wiederverwendbarer Sicherheitskomponenten sein. Ein anderer Grund liegt sicherlich darin, dass heutige Anwendungen unterschiedliche Anforderungen an die Sicherheit stellen, die oftmals nicht mit den gleichen Mechanismen realisiert werden können. All dies führt dazu, dass Sicherheit heute oftmals direkt im Code realisiert wird. Da der Code nun mit speziellen Sicherheitsanweisungen angereichert ist, die zum einen sowohl applikationsspezifisch, zum anderen aber auch kundenspezifisch sind, vermindert diese Methodik die Wiederverwendbarkeit, Wartbarkeit und Flexibilität des Codes. Gerade im Bereich der komponentenorientierten Softwareentwicklung sind diese Aspekte enorm wichtig weshalb die Sicherheitslogik strikt von der Anwendungslogik getrennt werden muss.

In dieser Dissertation wird GAMMA, ein plattform- und architekturneutrales Rahmenwerk vorgestellt, das wiederverwendbare Komponenten zur Authentifizierung, Autorisierung und Protokollierung anbietet. Alle Komponenten basieren auf *deklarativer Sicherheit* die eine vollständige Abkapselung von Anwendungs- und Sicherheitslogik erlaubt. Diese Abkapselung ermöglicht in weiterer Folge die Erstellung von hoch flexiblem, wiederverwendbarem und trotzdem sicherem Code. Die Realisierbarkeit dieses Rahmenwerks wird dann anhand zweier Referenzimplementierungen bewiesen, die zahlreiche wiederverwendbare Sicherheitskomponenten anbieten die direkt und intuitiv in Softwareprogramme integriert werden können.

Abstract

Security is nowadays recognized as an absolute need in software development. Although thoroughly researched concepts for access control exist that have been proven in mainframe computing, we still lack of adequate mechanisms that can be used in today's development of modern software architectures. However, currently we face the situation that security mechanisms have often been added to existing software causing many of the well-known deficiencies found in software products. One reason may be the lack of appropriate reusable components that support application developers. Another reason might be that applications have diverse security requirements that cannot be handled adequately. Thus, security is often addressed and implemented directly into the code, reducing reusability, maintainability, and flexibility aspects. However, with rise of component-based software development security models needs to be made available for reuse, encapsulating the security logic from the business logic.

This thesis presents GAMMA, a platform and architecture neutral framework, that offers reusable authentication, authorization, and auditing mechanisms by providing declarative security mechanisms. Declarative security allows the decoupling of security logic completely from the application logic, allowing to write highly flexible, reusable but still security aware software components and applications. Furthermore, this concept is proven by presenting a reference implementation of this framework which offers several ready-to-use but still extensible authentication, authorization, and auditing mechanisms that can be transparently integrated into applications.

Thanks and Acknowledgements

This thesis took a long time to write. Many people helped, both directly and indirectly for which I'm very thankful.

First and foremost I thank my mentor and internal supervisor *Dr. Wolfgang Essmayr* for his contribution towards my education and being a good discussion partner for various security aspects. Of course I want to say thank you to both of my academic supervisors, namely *a.Univ.Prof. Dr. Josef Küng*, and *a.Univ.Prof. Mag. Dr. Werner Retschitzegger* who provided helpful reviews and hints which helped me writing this thesis. Furthermore, I acknowledge *Dr. Dagmar Auer*, who provided me with very helpful feedback concerning the thesis but also new ideas concerning the work itself.

Additionally, I thank my research colleagues for great discussions and new ideas concerning the appliance of the GAMMA framework. Especially I want to mention *Thomas Ziebermayr* for discussing the idea of applying the framework to the Web Service technology, *Rudolf Ramler* for finding new ideas about how to test security and the resulting code, and *Mario Pichler* for evaluating the use of GAMMA in the area of ubiquitous computing or peer to peer networks.

This work would not have been possible without the support of the *Software Competence Center Hagenberg*, where I got the unique opportunity to participate in a strategic security research project and the SCCH PhD program.

Finally, I thank my family, especially my mother *Liane Probst*, who always supported me during my entire studies.

Table of Contents

1 Introduction

This work aims to present a framework that allows the transparent and active integration of reusable security components. Current available solutions require either extra effort when trying to secure an application (programmatic security) or address special target environments thus requiring a special underlying platform or architecture.

The work presented here tries to overcome existing shortcomings by providing a security environment that protects application objects. This environment is neatly integrated into the target application which enables the transparent use of the framework. In fact, the application developer does not have to put extra effort into application development since security is established outside the application.

1.1 Problem Statement

Security is an absolute need in today's software applications. Especially web-based or web-integrated software applications need sophisticated security mechanisms (compare to Kabay). Various security mechanisms and products are nowadays available enabling a secure communication via the Web. Although modern programming environments start to offer security mechanisms to protect the application itself, there is still a lack of transparently and efficiently supporting the developer to establish adequate security models, since existing solutions are most of the time not expressive enough or cannot be sufficiently adapted to complex application requirements. Furthermore, solutions provided from today's software environments require programmatic security instead of more flexible declarative security.

Experiences gained from software engineering show the need for reusable security components that can be neatly and transparently integrated into the software development process, beginning already at the early steps. This means that a set of security components is available, which can be first used in a generic way and then can be adapted to the evolving security requirements during any stage of the development cycle. In order to address various application domains, these

components should be architecture neutral, meaning that the concept itself can be realized in various programming environments and operating systems. Consequently, the mechanisms must be independent from the programming language, meaning that they must not rely on any special mechanism provided by a certain programming language (e.g. Java sandbox).

A lot of sophisticated security models have been published within the last years. Security models combine various mechanisms in order to enforce some security requirements, stated in a so-called *security policy*. However, today's software products have complex security requirements which often cannot be covered with a single security model. Although not supported by most systems, a combination of various models would help to solve many problems and address complex security requirements.

1.2 Goals

Looking at all requirements stated above, the aim of this work is to provide a concept that allows the easy integration of security components to software applications. The main goals are to provide:

- *A Platform and architecture neutral framework* that supports various kinds of application and target domains,
- *Active support for application developers* that allows an easy integration of security models into applications. This means that application developers are supported in all stages of software development by providing adequate security components that can be adapted to the current stage of the software life cycle.
- *Multiple or customized security models* to address any kind of security requirements, providing components for authentication, access control, and auditing.
- *Declarative security mechanisms* that decouple security from application logic on one side and allow the adaptation of the security components at any time on the other side. Declarative security prospects a maximum of flexibility and adaptability of the security models and mechanisms (see Probst and Küng, 2004). Thus, providing declarative security mechanisms is the most important goal of this work.

In order to meet all the goals presented above, the framework must be:

- *Generic*: being appropriate to arbitrary application domains.
- *Ready-to-use*: a set of state-of-the-art security models (e.g., RBAC, DAC) that can be directly and easily utilized requiring only minimal effort for modifications.
- *Expressive*: the security concepts implemented within the framework should be semantically rich, covering a range of improvements over the state of the art.
- *Adaptable*: the supplied security models can be tailored to specific requirements of the application domain for instance, multiple models may be combined simultaneously.
- *Extensible*: since security is a rapidly changing field in computer science, the framework must allow the integration of modern and new security concepts (e.g., biometric authentication). Ideally, these concepts can be introduced without the need for re-learning anything about the framework.
- *Enforceable*: the framework should meet all of these requirements in an efficient and reliable way contributing to the trustworthiness of (electronic) business applications.
- *Flexible*: security requirements often change during the lifetime of a software application. The framework should be flexible enough to address changes in the underlying security policy at any time of the software's lifecycle.

The practical relevance is shown by a reference implementation based on Java. Java has the advantage of being independent from the underlying hardware and operating system. Since the framework aims to be independent of a certain programming language, the feasibility of providing a reference implementation using the Microsoft .NET framework is also investigated. Although not a complete implementation is provided, this work shows how the core parts and the main features can be realized in .NET.

1.3 Basic Terminology

The following explains important terms which are essential to understand the framework, its motivation and idea.

1.3.1 Security Policy

Gollmann (1999) defines the term *security policy* as follows: A security policy is a set of rules that state which actions are permitted and which actions are prohibited. The domain of a security policy is the set of entities, i.e. users, data objects, machines, that are governed by the policy.

In other words, a security policy defines the boundaries and the security-relevant conditions of a software application.

1.3.2 Security Models

Castano et al. (1995) describe *security models* as follows: A security model provides a semantically rich representation in that it allows functional and structural properties of the security system to be described. A security model allows the developers to give a high-level definition of the protection requirements and system policies as well as producing a concise and precise description of the desired system behavior.

Another definition is taken from Gollmann (1999): A security model enables the formulation of a security policy by describing the entities governed by the policy and stating the rules that constitute the policy.

In our context, we can see a security model as a means that consists of various security mechanisms and states how these mechanisms have to be used in order to meet the requirements stated in the security policy.

1.3.3 Programmatic Security

Programmatic security means that the application developer addresses the security requirements directly in the code. Thus, the code is filled up with statements that verify the user's privileges. Programmatic security allows to address complex and very specific security requirements but since the code is popped up with security statements, the reusability of the code and the flexibility of the underlying security policy is decreased. Since security checks are hard-coded in the application logic, the resulting software has to be changed every time the security requirements change. In other words, security has to be programmed into the code which has negative impacts on reusability of the code on the one hand and results in a practice of permanently re-inventing the wheel on the other hand.

1.3.4 Declarative Security

Declarative security on the other hand defines the security policy outside the application's code. However, the privileges of a user depend on the current security policy. Since the code does not contain any application or domain-specific security code, the code can be reused more efficiently. Furthermore, if the security policy changes, only the policy file has to be modified – not the code. This allows more flexibility to changes, thus having a positive impact on maintenance aspects. Nevertheless, since declarative security is more general than programmatic security, it is harder to provide appropriate solutions that are adaptable to various security requirements and application domains.

1.3.5 Programmatic versus Declarative Security

The difference between programmatic and declarative security is best demonstrated by a simple example. Let us assume a company where certain financial aspects (e.g. settling accounts) are done by the secretary. Using programmatic security, the method that realizes the settling of accounts has a statement that says that only a user who acts as a secretary is allowed to execute the following code segment. In the case of declarative security, the policy file states that the method for settling accounts can only be called by the secretary. The environment ensures that this method can only be invoked if the calling user acts as a "secretary". The method itself does not perform any further security checks.

Both solutions are adequate and ensure that only a secretary can settle accounts. The difference becomes visible when the security requirements changes later on. Let us now assume that the company mandates a consultant who needs to proof the settling of accounts. Thus, also the consultant must be able to invoke the appropriate method. Using programmatic security, the code must be modified by allowing the function secretary and consultant, recompiled and retested. Declarative security requires only a simple entry in the policy file stating that secretary and consultants are allowed to call the method. No recompilation or modification in the application code is necessary.

This example shows on a small scale the advantages and importance of declarative security. Nonetheless, as mentioned above, such declarative security is hard to achieve since a special environment is needed that externally steers the application. In fact, there exists only few solutions that support such declarative security.

1.4 Structure

The remainder of this work is structured as follows:

Chapter 2 discusses the necessary basis for understanding the work presented in this thesis. In fact, security models are discussed which form the basis for developing security aware applications. Chapter 3 then presents GAMMA, a framework that actively supports software developers and architects in integrating security mechanisms into their applications. Furthermore, a reference implementation of GAMMA is presented, showing the feasibility of the framework. Chapter 4 discusses the work and compares the framework to existing solutions. Finally, Chapter 5 contains a concluding discussion.

2 State of the Art in Security Modeling

The work presented in this thesis aims to provide a framework for integrating various what security models represent or how they are used in today's software applications. This chapter introduces the modeling of security requirements and shows the practical relevance of security models. Furthermore, similar or related approaches to the aimed GAMMA framework are presented. Since GAMMA is designed to support developers in integrating security, a study of the offered security mechanisms of today's most frequently used development platforms is done. In a second step, related work with similar goals are presented and analyzed. The result of this study leads to the requirements that needs to be addressed in GAMMA. In order to compare GAMMA with existing solutions in Chapter 4, a criteria catalogue is presented.

2.1 The Security Levels

Before looking at security models and modeling in detail, an overview of the different security levels is given. As illustrated in Figure 1, there are several levels of security mechanisms (see also Probst et al., 2002). Depending on the degree of the integration of security mechanisms into the application, one decides between *lower-level security* mechanisms and *high-level security mechanisms*.

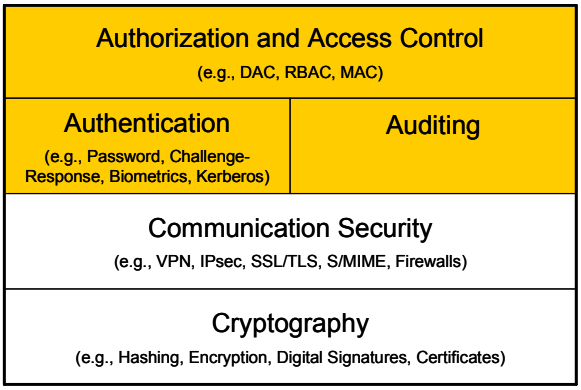


Figure 1: Levels of security mechanisms

Lower level security mechanisms are decoupled from the application itself and provide a common base for secure transmission of data or the secure usage of IT equipment. Lower level security mechanisms consists of two levels, the *Cryptography* and the *Communication Security* level. The cryptography level offers various – often mathematical – means and algorithms for realizing different security requirements like the calculation of one-way hash codes, chipper codes, digital signatures or certificates. The communication security level uses these offered algorithms to secure the communication between different IT components or the related data flow. Products and techniques like *Virtual Private Networks* (VPNs), *IPSec*, or *SSL* rely on cryptography algorithms in order to do their work.

High-level security relies on the mechanisms provided by the lower level security and establishes the possibility to secure software applications. The three main components are *Authentication*, *Auditing*, and *Authorization and Access Control*. According to Sandhu and Samarati (1996) these mechanisms are the core features of high-level security and form a security model. Especially in the field of lower level security, reusable components for the development of security-aware applications are available. At the higher levels adequate components often require a specific platform and/or architecture and are most of the time not expressive enough or too restrictive. Thus, this work concentrates on high-level security components and presents an architecture that allows the reuse of components at this level.

2.2 Security Modeling

As described above, security modeling combines the high-level security mechanisms in order to provide a secure environment in software applications. Before security modeling is explained in detail, some definitions of security modeling are given.

“The objective of security modeling is to produce a high-level, software independent conceptual model, starting with requirements specifications that describe what needs to be protected in a system. A security model describes functional and structural properties of the security system. At design time of an application, the security model allows developers to give a high-level definition of the protection requirements and system policies as well as a concise and precise description of the desired system behavior” (in Essmayr et al, 2004).

“If we assume the existence of a set of objects, which can be intuitively viewed as consisting of information receptacles, and a set of subjects, which can be intuitively

viewed as consisting of agents who can operate on objects in various ways, security is the problem of appropriately governing subjects' access to objects" (McLean, 1990).

Thus security models can be seen as a means of enforcement of the system wide security policy. In fact, they try to represent matter of facts or procedures known from the real world (e.g. the analogy of possessing something). Using such security models allows the application developer to address security requirements already at the early stages of the software development cycle.

2.2.1 Parts of a Security Model

Generally, a security model is stated in terms of *subjects*, *objects* and *authorizations* as shown in Figure 2.

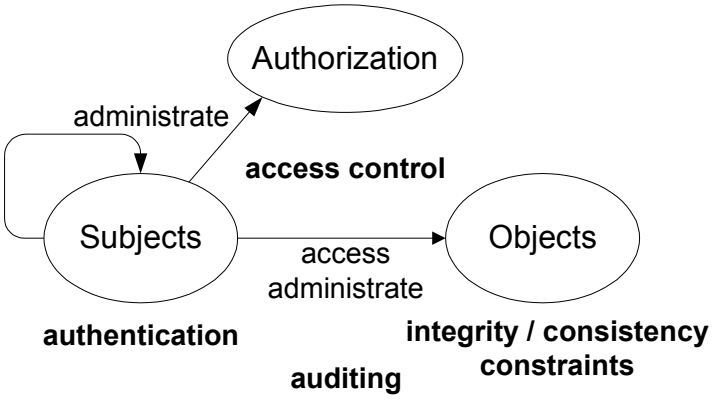


Figure 2: Generic security model

Subjects are the active entities within an application that are able to process data. These are not necessarily only users, since processes also perform access operations on objects. However, since processes are always acting on behalf of a user, the access privileges of a process should not exceed the privileges of its initiator. Objects are passive entities that contains data. This data has to be protected in order to enforce confidentiality. This protection is expressed by authorizations that state the actions a subject can perform on an object. Thus, each access limitation can be expressed as a tuple that consists of at least one subject, one object, and one authorization. However, in practice it is not as simple as it may look like. Not every object in an application domain has to be protected, so the security model must be aware of having protected and unprotected objects. Furthermore, in object-oriented environments subjects are often regarded as objects too, depending on the current view on the subject. On the other side, a passive object can become active when

calling certain processing methods and thus has to be regarded as subject too. All these issues must be addressed by the security model as well which makes the development and usage of such models sometimes a challenge.

2.2.2 Mechanisms in Security Models

A security model consists of several mechanisms to enforce the underlying security policy. These mechanisms were already presented at a glance above. The following contains a more detailed view on this mechanisms which is mostly taken from Sandhu and Samarati (1996).

- *Authentication* is the process of verifying a subject's identity. In particular, the authentication may be one of user-to-process or process-to-process authentication. User authentication mechanisms typically base the decision on something the user *knows* (e.g., password, PIN code), something the user *has* (e.g. private key on a smart card), or something the user *is* (e.g. biometric signature of a fingerprint) (see Essmayr et al, 2004). Thus, a security model is able to state which rules are obligatory and which can be omitted during the access checking mechanism. A subject is understood as an active entity within an application that is able to process data.
- *Authorization and Access Control* determines, if a certain subject has appropriate permissions to access an object. This determination is based on a system-wide security policy which is stated and enforced through the security model. An object is understood as a passive entity that contains data. This data has to be protected in order to enforce confidentiality.
- The *Auditing* mechanism gathers data about activities in the system in order to detect violations of the security policy or failures of the security model. The analysis of the resulting audit trail can be done offline at a later point in time or online in real-time. The latter allows a direct view on the system's state and allows the recognitions of deviations from the normal state immediately. These systems are known as *Intrusion Detection*.

The authentication mechanism thus forms the central part of the security model since all other mechanisms depend on it. Authorization is impossible without knowing the subject that wants to process a certain object. Furthermore, it is rather unmeaning to audit successful or failed access checks without knowing the initiator.

Depending on the target system (e.g. databases) further mechanisms are often included for covering special aspects of the target environment. In fact, integrity and

consistency constraints can also be regarded as a part of a security model. Since this work concentrates on authorization and access control aspects, such enhanced mechanisms are only mentioned when appropriate.

2.2.3 Classification of Security Models

Security models can be classified according two major categories, depending on how they handle authorization and how the administration is done (Fernandez et al., 1996).

The authorization handling of security models can be classified according to the following three approaches:

- *Authorization-rule based models*: These models express authorizations in form of rules specifying the type of access each subject has to a particular object. The minimal form of such a rule consists of a subject, an object and the authorized type of access. Depending on the idea behind the model, rules may contain additional information for administration purposes such as a grantor, an owner, or any other administrative predicate.
- *Multi-level models*: These models assign security levels to subjects and objects and regulate the information flow between these security levels. Multi-level models are very restrictive and secure but often difficult to apply in real-world applications since there is often no such clear notation of security levels.
- *Mixture of authorization-rule based and multi-level models*: Some models combine the authorization-rule based and multi-level approaches in order to enhance the capabilities of the security model. In fact, each security level then maintains authorization rules that additionally regulate the access of its subjects.

According to the administration aspect, security models can be classified into the following three categories:

- *Decentralized administrated models*: These models do not have a central manager, instead there are many subjects that are able to alter access rules. In fact, these models follow the *ownership paradigm*, saying that subjects own objects they have created. Thus the owner is the only entity that is able to state who else is allowed to use the object. Decentralized administration of authorization is rather flexible and takes care of particular requirements of individual subjects. However, since permissions can be passed on to third subjects, the problem of cascading and cyclic authorizations arises. Furthermore,

decentralized authorization may contradict the situation in many enterprises, where information is owned by the whole enterprise rather than by several individuals.

- *Centralized administrated models*: These models have a central manager who administrates subjects, objects, and the related authorizations. The advantage of these models is the easy administration and the clear view of the current authorization state within the model. Furthermore, the problem of cascading or cyclic authorizations does not exist. The disadvantage of these models is their inflexibility, since subjects have to negotiate with the central administration unit when they need access to particular objects.
- *Mixture of decentralized and centralized administrated models*: Some security models combine the two administration approaches and their advantages. In these models subjects are centrally authorized for a number of objects but still manage their own objects, which may be shared with other subjects.

The combination of the different authorization and administration approaches is illustrated in Figure 3. Some famous security models are placed within the dimensions described above in the diagram with respect to their authorization and administration approach. These security models are described later in this chapter in detail.

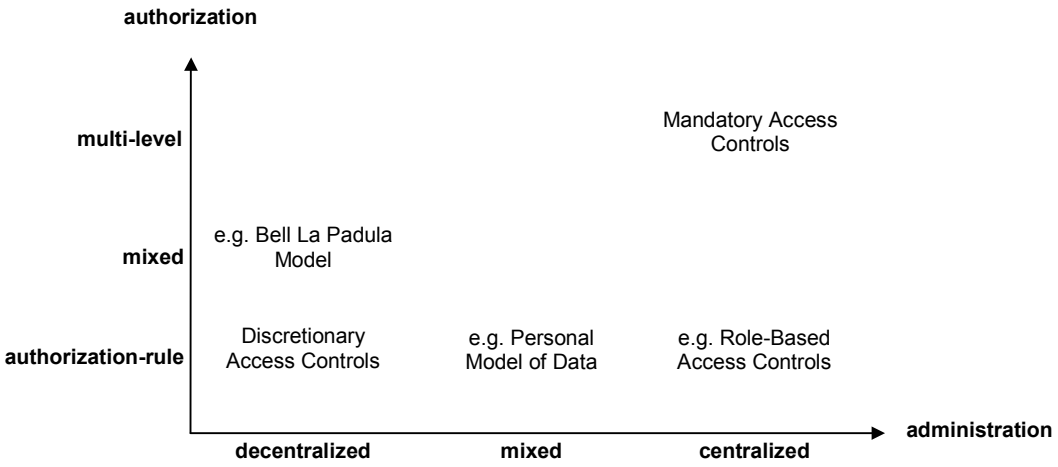


Figure 3: Different approaches to authorization and administration

2.2.4 Security Models at Work

Knowing the different approaches of security models, the question arises, how these models work and make their access decisions. It is understood that the decision

finding will vary from model to model depending on their approaches, but in principle they perform their work very similar. Each security model contains a decision base that is consulted when the model has to decide, whether to grant or deny a specific action. According to the entries in the decision base the model is able to make such decisions.

In the case of multilevel models, these entries state the allowed flow of information between the various security levels. In order to make a final decision, the security model must know the assigned security levels of the subject and the object. Knowing these levels, the model can consult its decision base and determine if the planned flow of information is conform to the model's policy.

In the case of authorization-rule based models the decision base consists of a set of authorization rules. Each rule consists at least of a subject, an object, and an authorization stating that the given subject has a specific authorization on an object. In order to make a final decision, the security model consults all fitting rules for a specific access request. If a matching rule is found, the model's decision is derived from the authorization. Thus, the question arises what to do if no appropriate rule is found, since it is nearly impossible to state a rule for each access situation. This problem is solved by specifying a *world assumption*, which can be either open or closed. An *open world assumption* means that everything is allowed by default and the model's rules restrict access to objects. Within the *closed world assumption* everything is denied by default whereas the model's rules grant access to objects. However, both assumptions have positive and negative authorizations in order to provide more flexibility. In those models the assumption only defines the default behavior specifying which access decision has to be made when no appropriate rule is found.

The most frequently used security models are *decentralized, authorization-rule* based models, which are commonly referred to as *discretionary access control* (DAC) models. In military environments, however, a *centralized, multi-level* approach is used, which is commonly referred to as *mandatory access controls* (MAC). Particular representatives of DAC and MAC models are roughly mentioned in the following subchapters. A detailed description can be found in Castano et al. (1995). *Centralized, authorization-rule* based security models have received new interest lately due to *role-based access control* (RBAC). Since the RBAC model is one of the most important security models, this model is discussed in more detail than DAC and MAC.

2.3 Discretionary Access Control (DAC) Models

Discretionary security models grant the access of users to data on the basis of the user's identity. Since this model is authorization-rule based, the decision base consists of a set of rules whereas the rules specify the types of access the user is allowed to activate for a certain object. Any request of the user for an object is checked against specified authorizations, which grant or deny access to the data. The possessor of an object has the overall control on his object and can exclusively decide who else can access the object.

Discretionary models generally allow a user to authorize access to the data to other users. The most common form of administration is the ownership paradigm where the creator of an object is allowed to grant or revoke access to his object. Discretionary models represent a flexible way to enforce different protection requirements. The following subchapters contain a list of existing DAC approaches.

2.3.1 Access Matrix Model

The access matrix model states the authorization base using a matrix correlating the subjects, objects and the authorizations owned by each subject on each object. An entry in the matrix contains the access modes of a subject on an object. The set of access modes depends on the type of the objects and the system functionalities. The authorization state can be modified by a set of commands. Commands are composed of a sequence of primitive operations that modify the access matrix.

2.3.2 Take-Grant Model

The Take-Grant model can be considered as an extension of the access matrix model. The difference lies in the representation of the access rules. In fact, authorizations in the system are represented by a graph structure, since this representation is more efficient and saves required memory space. The graph itself can be easily represented by its adjacency matrix whose entries represent the values of the arcs labels. The state of the system is described by a triple, containing a set of subjects, a set of objects and a graph describing the system's authorization state. The graph's nodes represent the subjects and objects of the system.

2.3.3 Acten Model

The Acten model is based on the Take-Grant model but includes further administrative privileges and predicates on authorizations. Thus the Acten model allows a strict separation between administration and access control.

2.3.4 Wood et al. Model

The Wood et al. model considers the three-level architecture of the ANSI/SPARC proposal for databases which separates the database into three levels:

- *External level*: the user's view on the database
- *Conceptual level*: representation of the data stored in the database
- *Internal level*: physical storing of the database

Although there are three levels, the Wood et al. model only considers a relational model on the external level and an entity-relationship model on the conceptual level. The model treats the problems of authorization at different levels, of inter-level consistency as well as the issue of access decisions. Subjects are categorized into authorizers, who administer the authorizations, and the users, who access data according to the authorizations specified by the authorizers.

The model assumes that any operation on an external object can be mapped to a set of operations on one or more conceptual objects. Therefore a mapping function has to be defined for each table in the system.

Authorizations are described by access rules, which state that a subject can exercise a specific access mode on an object under conditions expressed by predicates. An authorizer, who is in charge of applying the security policies of the organization, defines access rules. The model considers a closed world assumption. Absence of a rule granting a subject access to a conceptual object implies that this subject has no access to any of the occurrences of that object through any external view.

2.4 Mandatory Access Control (MAC) Models

Mandatory security models grant access to data by the individuals on the basis of the classifications of subjects and objects in the system. Thus, each subject and object must be categorized to a certain security level. In fact, each subject and object is marked with a security stamp defining the current security level. The decision base of

the model consists of several mathematical relations between subjects and objects on the basis of the security level classification. Access is only granted if the appropriate mathematical relations between the subject and the object are fulfilled.

The following subchapters mention some concrete implementations of MAC models. Since MAC models are very restrictive, they are often combined with other approaches (e.g. DAC). If such combinations exist, the combination and the resulting advantage is also mentioned.

2.4.1 Bell-LaPadula Model

The idea of the Bell-LaPadula model was born during the early stages of multi-user operating systems. The model is dedicated to protecting the secrecy of objects in restricting the flow of information within a lattice of security levels. In order to do this, the model classifies objects and subjects (MAC) on the one hand and on the other hand it uses an additional access control matrix (DAC) since the classification is to restrictive in the field of operating systems.

In its core functionality, the model categorizes each subject and object into various security levels. The model restricts the flow of information between security levels by allowing only certain directions of information flow. The security model controls each access of a subject on an object. If the resulting flow violates the security policy stated in the model's decision base, the model intercepts the information flow and the access is denied.

In principle, the model allows information flow from only between similar levels or from lower levels to higher levels. This avoids unwanted disclosure of classified information. However, for flexibility purposes the model allows that subjects can change their level temporarily between predefined borders.

2.4.2 Biba Model

The Biba model extends the Bell-LaPadula model by introducing integrity functions. Subjects and objects are now categorized into integrity levels. The information flow is only allowed from higher levels to lower levels (no write up) which is regulated through static integrity rules. These integrity rules also address aspects like unauthorized copying of information. If a subject opens an object A for reading purposes, the subject can write to another object B only if object A resides in the same or in a higher integrity level than object B.

For more flexibility, dynamic integrity rules allow subjects to adjust their integrity level within a previously defined border. This enables a subject to deliver a message to a higher classified subject.

Another interesting point is the support of delegation of rights which is often required by operating systems. The Biba model treats users as well as processes as subjects resulting in the scenario that subjects (processes) can act on behalf of other subjects (users).

2.4.3 Dion Model

The Dion model proposes a clear mandatory policy, which protects the secrecy as well as the integrity of data. Basically it combines the principles for controlling secrecy of the Bell-LaPadula model with the principles of the strict integrity of the Biba model. The most significant characteristic is that the information flow between subjects and objects is not allowed. Flow of information is only allowed between objects whereas a subject's task is to establish connections between several objects. The security model verifies if a subject is allowed to establish a connection between two objects or not. This decision is based on the security levels of the subject and the participating objects.

2.4.4 The Sea View Model

The Secure Data View (Sea View) model grants access to the data stored in a database on the basis of mandatory as well as discretionary policies. It is formulated in two layers: the MAC (Mandatory Access Control) model and the TCB (Trusted Computing Base) model. The TCB model is layered on top of the MAC model. In fact, all the information of the TCB is stored in objects mediated by the MAC reference monitor.

2.4.5 The Jajodia and Sandhu Model

The Jajodia and Sandhu model is based on the security classifications introduced in the Bell-LaPadula model. Since this model deals with multilevel data, it extends the standard relational model to include classification labels and poly-instantiation. This allows that a stored object can be assigned multiple times to different security levels.

2.4.6 Smith and Winslett Model

The Smith and Winslett model is based on the concept of *belief*. This model assumes that a database is a set of ordinary relation databases, one database for each level in the security lattice. The database at a given level contains the total beliefs of the subjects of that level and reflects the state of the world in the schema. A subject believes the contents of the database at its own level and sees what it and the subjects at lower levels believe. Of course, believe conditions have to be integrated into operations of subjects on objects.

2.4.7 The Lattice Model for Flow Control

All models presented by now are limiting access according the direction of information flow by allowing the flow of information only in specified directions. The Lattice Model for Flow Control undermine this strict direction approach by introducing access rights that represent information flows and their directions. The model bases on a mathematical structure which formulates the requirements needed for secure information flows. This is done by categorizing objects into classes. The model itself defines secure routes between those classes. During the access checking mechanism, the object validates if access is done using secure routes.

2.5 Role-based Access Control (RBAC) Models

During the study it was pointed out that the role-based access control (RBAC) model seems to be the most promising approach for addressing security needs in today's software applications (see Sandhu et al, 2000; Sandhu 1996). Thus, this subchapter gives a detailed and thoroughly description of RBAC, its use, advantages, and disadvantages.

RBAC is one of the most flexible security models with respect to authorization and can already be found in modern information and operating systems (e.g., Windows NT, Novell Netware, Oracle DBMS). The two major advantages of RBAC are:

- RBAC clearly separates between the questions *what has to be done* and *who has to do it* in that is assigns users to roles and defines permissions on roles, which in turn take effect when users activate the corresponding role.
- RBAC allows a clear distinction of the object model of an application from the subject and authorization model. The object model gives a view on objects to protect (e.g., tables, columns, entities), the subject model provides an image of

the active entities within a system (e.g., users, processes), and the authorization model describes how access between subjects and objects and the administration of it are regulated.

2.5.1 RBAC at a Glance

In principle, permissions are not assigned to users anymore but to roles which can be seen as functions or tasks within the software system. On one side, this has a positive effect on the administration, since often a task within a system is performed by multiple persons which need the same set of permissions. On the other side the administration is relieved when an individual changes his position within the enterprise and performs other tasks since the administration just has to modify the person-role assignment and not the more complex person-permission assignments. Another point is that the grouping of individuals with the same tasks into roles result in a smaller decision base which in fact increases performance during access checking. In comparison to the access matrix model, there has to be a single rule for each person within a department of an organization although these persons are performing the same tasks and thus need to have the same permissions. Of course, the rule base can become very large depending on the size of the department, having a lot of rules which differ only by stating different subjects. Using RBAC, such a rule base consists only of a single rule stating that members of the department have the required privileges.

2.5.2 What is RBAC?

RBAC is a proven technology for large-scale authorization that reduces the costs of administering access control policies, as well as making the process less error-prone (compare to Sandhu et al., 2000; Gavrila and Barkley, 1998). With RBAC, system administrators create roles according to the job functions performed in a company or organization, grant permissions to those roles and then assign users to the roles on the basis of their specific job responsibilities and qualifications. Thus, roles define the authority of users, the competence that users have and the trust that the company gives to the users. Roles define both, the specific individuals allowed to access objects and to which extend or in which mode they are allowed to access the object (see Sandhu and Coyne, 1996). Access decisions are based on the roles a user has activated (compare to Sandhu et al., 2000).

There are three important relationships used in RBAC: the user / role relationship, the role / role relationship and the role / permission relationship as shown in Figure 4. Within the user / role relationship, the user is assigned to a role and is granted the permissions defined in the role. The role / role relationship allows to build role-hierarchies by assigning a role A to another role B. After this assignment, B combines the permissions of A and B. The role / permission relationship assigns roles to permissions, which basically consist of an object and a corresponding access mode. The particular structure of permissions depends on the object model of the application domain.

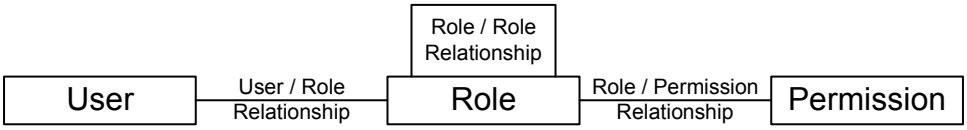


Figure 4: Relationships used in RBAC

RBAC also supports *separation of duties*, which provides the administrator with enhanced capabilities to specify and enforce enterprise policies compared to existing access control standards (see Sandhu et al., 2000).

Currently, considerable effort is conducted in order to standardize RBAC characteristics and concepts. The NIST model of RBAC (shown in Sandhu et al. 2000) is organized into four levels of increasing functional capabilities called flat RBAC, hierarchical RBAC, constrained RBAC and symmetric RBAC. These models are described in detail later. For a formal specification of RBAC please refer to Gavrila and Barkley (1998).

2.5.3 Characteristics of RBAC

In the following the characteristics of RBAC concerning its properties and administration issues are described.

2.5.3.1 Entities of RBAC

The basic RBAC model consists of four entities: users, roles, permissions, and sessions.

Users and Roles: A user is a subject, which is accessing different, protected objects. A role is a named job function that describes the authority, trust, responsibility, and competence of a role member.

Permissions: A permission is an approval of a particular mode of access to one or more objects in the system. Permissions describe which actions can be done on a protected object. Permissions can apply to single objects or to many. Both, permissions and users are assigned to roles. These assignments in turn define the scope of access rights a user has with respect to an object. Per definition, the user assignment and permission assignment relations are many-to-many relationships.

Sessions: Users establish sessions during which they may activate a subset of the roles they belong to. A session maps one user to possibly many roles, which results in the fact that multiple roles can be activated simultaneously and every session is assigned with a single user. A user might have multiple sessions opened simultaneously. A user belonging to several roles can invoke any subset of them that enables tasks to be accomplished in a session. In other words, sessions allow a dynamic activation of user privileges (see Sandhu and Coyne, 1996).

2.5.3.2 RBAC Properties

Sandhu et al. (2000) point out various properties of the NIST's RBAC model.

Scalability: The notion of scalability is multi-dimensional. RBAC provides scalability with respect to the number of roles, number of permissions, size of the role hierarchy, limits on user-role assignments, etc.

Authentication: RBAC *does not* address the issue of authentication. This issue is outside the scope of an access control model and is part of the system architecture, although authentication is absolute necessary for an appropriate work of an access control model.

Negative authorization: RBAC is based on *permissions* that confer the ability to do something on holders of the permission. RBAC *does not* contain negative authorizations (prohibitions), which deny access allowing to specify exceptions to the regular case.

Nature of permissions: The nature of permissions is *not* specified in the RBAC model. Permissions can be fine-grained or coarse-grained. Permissions can also be customized. The exact nature of permissions is determined by the nature of the application product.

Role activation: RBAC *does not* specify the ability of a user to select which roles are activated in a particular session. The only requirement is that it should be

possible to allow a user to activate multiple roles simultaneously. It does not matter if the user is able to activate explicitly roles or if all roles are automatically activated by the system. The type of activation is scope of the system's vendor.

Role engineering: RBAC *does not* provide guidelines for designing roles and assigning permissions and users to roles. This activity is called *role engineering*. This issue is outside the scope of RBAC.

Role revocation: Neither the semantics of role revocation in the RBAC model, nor a specified revocation behavior is defined. However, this is an important issue to which vendors and users of RBAC products must pay careful attention.

2.5.3.3 RBAC Constraints

Since permissions are organized into tasks by using roles, conflicts of interests are more evident than if dealing with permissions on an individual basis. In fact, a conflict of interest among permissions on an individual basis is hard if not impossible to determine at all. Using separation of duties among roles (i.e., defining mutually exclusive roles) provides the administrator with enhanced capabilities to specify and enforce enterprise policies. Since RBAC has static (user-role membership) and dynamic (role activation) aspects, the following two possibilities can be distinguished accordingly:

- *Static Separation of Duties (SSD):* is based on user-role membership. SSD enforces constraints on the assignment of users to roles. This means that if a user is authorized as a member of one role, the user is prohibited from being a member of a second role. Constraints are inherited within a role hierarchy.
- *Dynamic Separation of Duties (DSD):* is based on role activation. DSD is used, when a user is authorized for roles which must not be activated simultaneously. DSD is necessary to prohibit a user to undergo a policy requirement by activating another role.

Additionally to mutual exclusion constraints there may be further constraints specified for RBAC environments, which are not yet standardized within the NIST reference models.

- *Cardinality / Conditionality constraints:* defining the minimum or maximum number of users that may or must be assigned to particular roles respectively may or must activate particular roles so that the corresponding permissions take effect. For instance, think of a role "board member" that requires a minimum number of

four members from which at least two have to be present (i.e. activated the role) in order to make a decision.

- *Time constraints:* defining the periods of time when users may activate particular roles. For instance, it may be feasible that an employee may activate those roles which enable access to company relevant information only within the office hours.
- *Location constraints:* defining the physical places (i.e. IP addresses or domain names) from which particular roles may be activated. For instance, particular system administration tasks may be required to be done at a particular computer that is physically better protected than ordinary workstations.
- *History constraints:* defining constraints that relate to states of the past. Such kinds of constraints allow specifying a sequence of role activations or user-role membership (e.g. becoming a project manager requires to be senior software engineer).

Logically, any of these constraints may be combined with others within an RBAC environment though there is a tradeoff between the flexibility achieved with these constraints and the complexity to administrate and understand the effective authorization state.

2.5.3.4 Administration of RBAC

The administrative task of RBAC can be summarized as follows:

- *Defining roles* and organizing them within a hierarchy. For a detailed discussion of adding, removing or maintaining roles please refer to Nyanchama and Osborn (1994), who introduce a role graph to facilitate role administration.
- *Defining constraints* like static or dynamic separation of duties.
- *Assigning permissions* to roles. The structure of permissions depends on the object model that should be protected.
- *Granting and revoking membership* to the set of specified named roles within the system (see Ferraiolo and Kuhn, 1992). When a new employee enters the company, the administrator simply adds this person to one or more existing roles according to the users tasks and needs. Similarly, the users can be removed from a role when they leave the company or added to new roles when their function changes.

One of RBAC's biggest advantages is its easy administration, but managing a large number of roles can still be a difficult task. Sandhu and Coyne (1996) show how

RBAC might be used to manage itself. An administrative role hierarchy is introduced, which is mapped to a subset of the role hierarchy it manages.

2.5.3.5 Coexistence with MAC / DAC

Mandatory Access Control (MAC) controls access on the basis of security levels to which subjects an object is assigned. Discretionary Access Control (DAC) controls access to an object on the basis of an individual's permissions and / or prohibitions. RBAC is an independent component of these access controls, but can coexist with MAC and DAC if desired. In such a case, access is only allowed if permitted by RBAC, MAC and DAC. However, RBAC is more general than MAC or DAC. In any case, RBAC can be used to enforce MAC and DAC policies as shown in Osborn et al. (2000). The authors point out the possibilities and configurations necessary to use RBAC in the sense of MAC or DAC.

2.5.4 Reasons to use RBAC

Nowadays, the trend is to use application-independent facilities to support many applications with minimal customization. RBAC is providing such facilities. Moreover, sophisticated versions of RBAC include the capability to establish relations between roles, between permissions and roles, and between users and roles. For example, two roles can be established as mutual exclusive – the same user is not allowed to activate both. Roles can also acquire inheritance relations, whereby one role inherits permissions assigned to a parent role.

A study made at NIST (see Ferraiolo et al., 1993) indicates that permissions assigned to roles, unlike user membership in roles, tend to change relatively slow. Because RBAC is able to predefine role-permission relationships, it makes it simple to assign users to predefined roles.

Access control policy is embodied in RBAC components such as role-permission, user-role and role-role relationships. RBAC components can be configured directly by the system administrator or indirectly by appropriate roles as delegated by the system administrator. Because the access control policy can change over the system life cycle, RBAC offers a key benefit through its ability to modify access control to meet changing organizational needs. Because RBAC is not the overall solution (RBAC does not attempt to directly control the permissions for an event sequence within a workflow) other forms of access control can be layered on top of RBAC to extend its facilities.

2.5.5 The NIST Model for RBAC

In Sandhu et al. (2000), a unified standard in RBAC is presented: the so-called *NIST Model for RBAC*. This model is organized into four levels of increasing functional capabilities, which are flat RBAC, hierarchical RBAC, constrained RBAC, and symmetric RBAC. These levels are cumulative – each level adds exactly one new requirement. In the following the four levels are briefly presented.

2.5.5.1 Flat RBAC

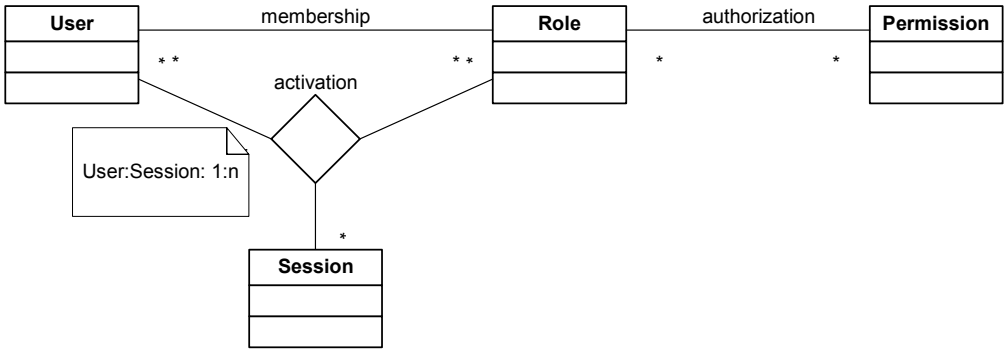


Figure 5: Flat RBAC

Figure 5 shows flat RBAC, which defines the essential aspects of RBAC. The basic principle is that users are assigned to roles (user/role assignment, indicated through the membership association), permissions are assigned to roles (permission/role assignment, indicated through the authorization association) and users gain permissions defined in the role(s) they activate. A user can activate several roles within a session (indicated through the n-ary activation association). It is required that these assignments are many-to-many relationships. This results in the fact that a user can be assigned to many roles and a role can contain several permissions. Flat RBAC requires a user/role review whereas the roles assigned to a specific user can be determined as well as users assigned to a specific role. Similarly, flat RBAC requires a permission/role review. Finally, flat RBAC requires that users can simultaneously exercise permissions of multiple roles they belong to.

Flat RBAC represents the traditional group-based access control as it can be found in various operating systems (e.g., Novell Netware, Windows NT). The requirements of flat RBAC are obvious and obligatory for any form of RBAC. According to Sandhu et al. (2000), the main issue in defining flat RBAC is to determine which features to exclude.

2.5.5.2 Hierarchical RBAC

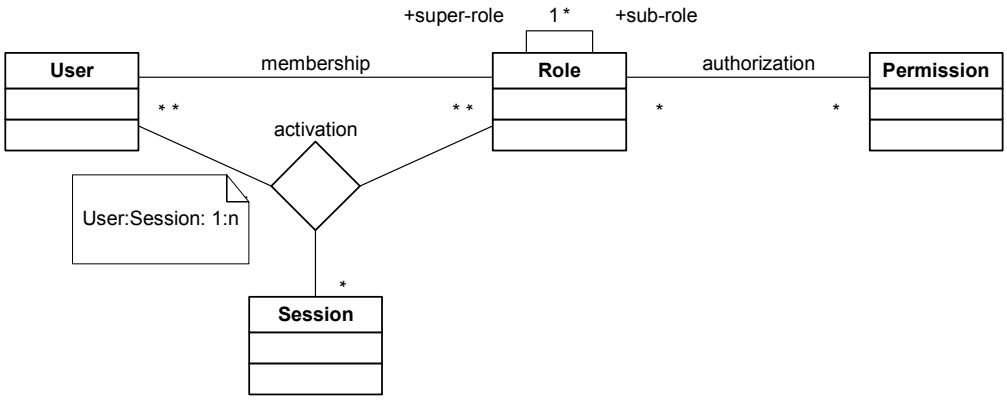


Figure 6: Hierarchical RBAC

Hierarchical RBAC supports role hierarchies. This is shown by the sub-role / super-role association in Figure 6. A hierarchy defines a seniority relation between roles, whereas senior roles acquire the permissions of their juniors. In fact, the NIST model recognizes two sub-levels:

- *General Hierarchical RBAC:* provides support for an arbitrary partial order to serve as the role hierarchy.
- *Restricted Hierarchical RBAC:* provides restrictions on the role hierarchy. Hierarchies are limited to simple structures such as trees or inverted trees.

Role hierarchies can be:

- *Inheritance hierarchies* whereby activation of a role implies activation of all junior roles,
- *Activation hierarchies* whereby without the implication of the activation of all junior roles (each junior role must be explicitly activated to enable its permissions in a session) or
- A *mixture* of both.

2.5.5.3 Constrained RBAC

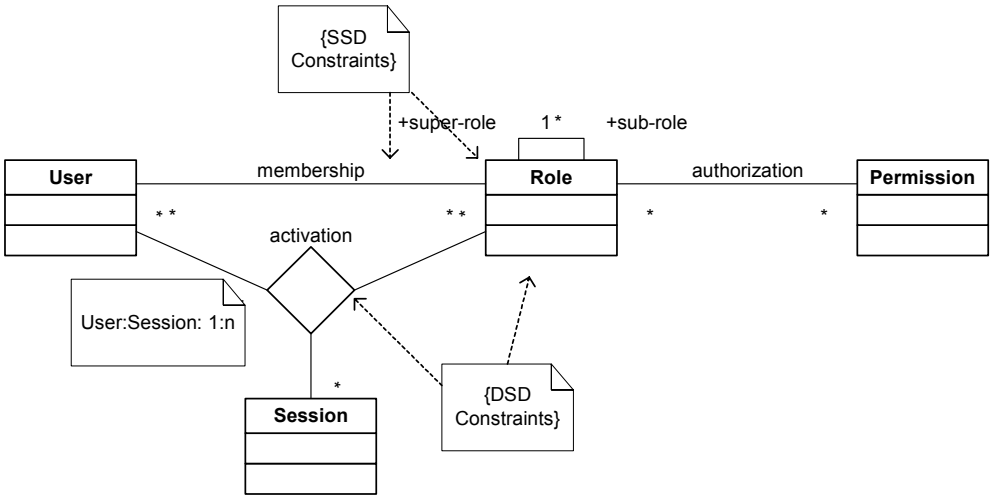


Figure 7: Constrained RBAC

Figure 7 shows a diagram of constrained RBAC. Within constrained RBAC there is the possibility of separation of duties (SOD). SOD is a technique for reducing the possibility of fraud and accidental damage. It spreads responsibility and authority for an action or task over multiple users thereby the risk of committing a fraudulent act by an individual is reduced. Many different SOD requirements have been identified. The most important ones have already been described in Chapter 2.5.3. In Figure 7, static separation of duties is shown through the SSD constraints and dynamic separation of duties is shown through the DSD constraints.

Sophisticated access control products should support constrained RBAC since it provides very useful methods for implementing the system's policy.

2.5.5.4 Symmetric RBAC

Symmetric RBAC adds requirements for permission/role review similar to the user/role review introduced in flat RBAC. Thus the roles to which a particular permission is assigned can be determined as well as permissions assigned to a specific role. The implementation of this requirement is rather difficult in large-scale distributed systems, thus it is seldom integrated in today's software applications.

2.5.6 Discussion of RBAC

In the following the advantages and disadvantages of the RBAC model are discussed.

2.5.6.1 Advantages of RBAC

One of RBAC's biggest advantage is the easy administration of RBAC since there is a strict separation between users, functions (roles) and permissions. This simplifies the permission management since the assignment of permissions to roles or functions within the enterprise is done in a natural way. RBAC's notation of roles is done in an enterprise or organizational concept. It reflects the view of permissions from the enterprise perspective.

RBAC provides a solid base with a rich set of functions for the enforcement of policies. This makes RBAC more scalable than other security models. Furthermore, RBAC is policy-neutral which means that RBAC is a means for articulating policies rather than embodying a particular security policy. The policy enforced in a system is a result of the precise configuration and interaction of RBAC components.

Finally, there exists a standardization for RBAC (compare the NIST model for RBAC, Sandhu et al., 2000).

2.5.6.2 Shortcomings of RBAC

On the other hand RBAC has some disadvantages. There are a lot of restrictions, thus not every security policy can be enforced using RBAC. A lot of needs resulted in several extensions as presented in Chapter 2.6.

RABC asserts that it reflects the structure of an enterprise within its security model. In fact, modern enterprises are often organized in sub-structures like departments, projects, etc. that cannot yet be handled adequately by the available concepts for role-hierarchies. Other extensions or concepts (e.g. role-templates) are necessary to handle such structures.

The relationship between discretionary and role-based models is not well understood, yet. A lot of RBAC implementations allow for posing permissions on roles and users as well, which contradicts the pure RBAC requirements. It is feasible to argue that a clear concept for the coexistence of discretionary and role-based aspects within one security model is needed for the near future in order to prevent further impure RBAC implementations. On the other hand it has been shown that RBAC without DAC features is too restrictive in order to meet today's security requirements.

Role-hierarchies are a natural part of advanced RBAC models. However, the relationships between roles may be much more complex than simple generalization

hierarchies. Roles may also participate in *part-of* relationships, which effectively leads to *virtual* roles (i.e. roles used for structuring cannot be activated directly by individuals). These virtual roles are also called *tasks* and received considerable research interest from workflow management groups (e.g. Castano and Fugini, 1998). Regarding virtual roles from the direction of the role/permission-relationship (i.e. from the permission side) virtual roles can be also called *named permissions*.

Furthermore, special constraints for the user-role membership are feasible thinking of *default roles* that are automatically activated if the user does not explicitly activate particular roles or *proxy membership*, which allows particular users to activate a role if other users are temporarily not available (e.g., system administrator). Within an increasingly mobile and networked world the definition of "socially established" roles (e.g., medical doctor, lawyer, professor) would be a substantial benefit. Those kinds of roles allow the specification of special permissions to a priori unknown individuals. Herzberg et al. (2000) and Opplinger et al. (2000) provide first concepts contributing to that idea.

Finally, modern software development is done in an object-oriented and component-based way. Today's software development has to deal with complex objects and subjects in distributed environments. RBAC lacks supporting these technologies, which also raises the need for extended RBAC models (e.g., OOAC, OBBAC).

2.6 RBAC-related Models

Since RBAC alone is not able to deal with all requirements of modern software applications, other RBAC related models and extensions have gained interests. Some of them are presented in this chapter.

2.6.1 Role Templates

Enterprises often entail recurring sub-structures like departments or projects. Using RBAC, these sub-structures are mapped to roles. However, roles are not adequately enough since each time a new instance of the structure is created (e.g. a new project starts), the whole role-administration has to be done (define a new role, assign members, assign permissions). Essmayr et al. (1998) propose an extension to the RBAC model which addresses this administration effort for recurring role-hierarchies, called *role templates*.

Role templates define a general hierarchy for recurring sub-structures. When a role template is instantiated it produces concrete roles within the role hierarchy with unique names.

The advantage of this approach is the reduction of administrational effort. Furthermore, instantiating a role template is less vulnerable to errors and can be done automatically.

Giuri and Igilo (1997) use role templates for providing content-based access controls. The authors describe a model that provides special mechanisms for the definition of content-based access control policies. This is done by using *parameterized privileges* to restrict the access on subsets of objects, and the concept of role templates to support composition and encapsulation of parameterized privileges. Such parameterized privileges are restricted privileges that contain a set of unbound variables. However, these privileges can be used by the access control system only if they are fully specified.

2.6.2 Team-based Access Control (TMAC)

Thomas (1997) presents TMAC, an approach to apply RBAC in collaborative environments. A team is understood as a collection of users in specific roles with the objective of accomplishing a specific task or goal. The motivation of TMAC is the need for a hybrid access control model that incorporates the advantages of broad role-based permissions across object types, context recognition associated with collaborative tasks, and the ability to apply this context to decisions for permission activation. Thus TMAC supports role-based, scalable permission assignment but also fine-grained, runtime permission activation at the level of individual users and objects. The basic idea in TMAC is to use RBAC to define a set of permissions across the domains of the total set of roles in the information system and the set of object types. The approach is well summarized in Essmayr et al. (Essmayr, 2004).

2.6.3 Role- and Task-based Model (R&T model)

The R&T model (compare Schier, 1998) aims to bind permissions to tasks instead of roles. Tasks can be performed by different roles. Roles are also authorized for subjects and describe a structure for actions executing procedures on defined objects. The basic idea is to provide an extra dynamic aspect during access checking. Within RBAC, dynamic separation of duties means that a user cannot activate two conflicting roles at the same time. However, the R&T model addresses the issue that

two roles are only conflicting in certain tasks, thus separation of duties should also consider the current context of the user. For example, a bank clerk can open the accounts of two competitors only if he performs statistical analyses (task 1) but not when performing comparisons between the results of the companies (task 2). Thus, a conflict of interest between the two roles arises only when the clerk is performing the second task but not within the first one.

The R&T model provides administrators with the capability to define who can perform what kind of actions, when and on which objects. Furthermore, the model itself is more dynamical by taking the current task of the user into consideration when making access decisions.

2.6.4 Object-Based Access Control (OBBAC)

Since today's software development is done in an object or component oriented way, there is also a need for object oriented concepts in authorization models (Izaki et al, 2000). The OBBAC model (Tenday et al., 1999) presents a way to specify an RBAC-conform security policy in an object oriented way, addressing object oriented concepts.

Technically, in an object oriented system a subject delivers a message to an object that processes this message and modifies its state or other objects. Thus an object can become a subject and the other way round.

The OOBAC model is based on the association of security labels to subject and object operations. The model allows then the operation according to the value of the security label of the calling subject.

2.6.5 Object-Oriented Access Control (OOAC)

Another approach to support object-oriented concepts in RBAC is presented in Essmayr et al. (1997). The authors show the integration of access control mechanisms in object-oriented and relational databases.

The presented OOAC model intercepts messages sent from the subject to the object and validates this message if it is conform to the security policy. If so, the message is delivered to the target object and can be processed. If denied, the message is blocked and an access control exception is raised.

2.7 Other Models

This subchapter presents other existing models that do not fit into one of the approaches presented above (MAC, DAC, RBAC). However, most of these models follow a certain purpose rendering them useable only for specific applications or security requirements.

2.7.1 Chinese Wall

The Chinese Wall model addresses conflicts of interest when accessing data. The model considers existing access rules of certain subjects and restricts access to other objects according to these rules. The model follows the idea that users build up a wall around the data-objects they need, thus denying access requests that cause conflicts. A good example is a consulting company. If this company consults an enterprise A and an enterprise B whereas A is a direct competitor of B, no member of the consulting company is allowed to consult A and B because of the danger of not-allowed information transfer.

Related objects (e.g. data of a certain company) are grouped into so-called company datasets. Furthermore, conflict classes are maintained that store conflicting datasets. An important point is that not only the current state of the system is relevant for decisions but also actions performed in the past (history).

Access to an object is only allowed if the object's dataset belongs the subject or if the subject is not member of any conflict class mapped to the object (direct conflicts). Additionally, the access control system checks, if the accessing subject has not accessed another object from a conflicting dataset in the past (indirect conflicts). Dynamic rules address the problem of accessing data via a third party. For example, it is possible that two competitors have their accounts at the same bank. The bank clerk has access to both data. The model allows access to an object only if no other object can be read that is related to a conflicting dataset.

The properties of this model require the evaluation of each access check dynamically. This means that the access control engine must check each relation of the subject and object and must consider and resolve the history of the subject which has negative influence on the performance of this model.

2.7.2 Personal Knowledge Approach

The personal knowledge approach (Pernul, 1994) focuses on protecting the privacy of individuals by restricting access to personal information in order to ensure the right of individuals. Privacy is understood as the right of an individual to choose which elements of his personal life may be disclosed. In the personal knowledge approach users and security objects are represented by an encapsulated person-object. The data part of this object corresponds to the individual's knowledge and the relationship to other persons whereas the operation part of the object corresponds to the possible actions which an individual may perform.

The problem of this approach is the limited usage since it addresses only the protection of an individual's data.

2.7.3 Clark and Wilson

The Clark and Wilson model addresses the integrity of data processed by applications. The model defines a set of transactions that manipulates data. These transactions are implemented in programs. The decision base of this model states that subjects are allowed to invoke certain transactions, thus the subject can invoke transactions but never get direct access to the object. If the subject needs the data stored in an object, it must access this data via a specific transaction. In fact, subjects are assigned to roles. Based on this assignment, users have to perform certain business functions which are mapped to database functions. It is therefore essential to state which user is acting in which role at what time and what transactions have to be carried out (Pernul, 1994).

The advantage lies in the fact that the model does not need to know the protected data itself since the data is abstracted by access transactions. The model itself considers data as trustworthy but does not trust the user. Thus the main problem of this model is the transfer from non trusted user input to trusted data. This is done by verification and transformation processes which are realized outside the model and are thus the weak point of this security model.

2.8 Security Modeling in Practice

Most of the presented security models are not fully applicable in practice. Often mixtures and combinations of the presented security models are used in order to cover complex security requirements or to provide more flexibility. Thus, this

subchapter discusses reasons why there is a gap between theory and practice and then shows how various security models are used in common software applications and their architectures.

2.8.1 Theory versus Practice

Security models in practice often differ from theory as they need to be less restrictive or more flexible. The RBAC model for example enables the assignment of authorizations to roles but not to users. Sandhu and Ahn (1998) describe the integration of RBAC in operating systems like Windows or UNIX. User-groups are described as the concept of roles. Both systems additionally allow the assignment of authorizations to users since only the assignment to groups (or roles) would be too restrictive.

Another reason is that software systems must often stay compatible to previous versions. RBAC has four levels whereas practical implementations often only support the first level. Again, Windows 2000 is a good example for this. After the installation of *Active Directory*[1] Windows is operating in the so called *mixed mode* which guarantees compatibility to older Windows NT domain controllers. Within this mode, only flat RBAC is supported with simple user groups and users to which permissions can be assigned. After switching to *native mode*, group hierarchies are supported too, which can be seen as hierarchical RBAC. Switching to this mode means loosing compatibility to older versions of Windows.

Furthermore, the complexity of some security models e.g., symmetric RBAC, or the lattice model for flow control are hard to realize and thus would result in high costs. Therefore, it is a common practice to choose easier models or to skip some model features.

The following contains a study of the integration of various security models into today's existing software environments and applications.

2.8.2 Microsoft Windows (2000, XP)

The operating system Windows realizes a combination of DAC and RBAC mechanisms. Each resource within the system is assigned to a user, the owner of the resource. The owner is allowed to control access to his resource and to pass access

---

[1] Windows becomes a domain controller after the installation of Active Directory.

rights to other users within the system (see Figure 8). The ownership can be taken by an authorized person such as the system administrator. Taking ownership is also an authorization that can be given to any user of the system. These features are clearly DAC mechanisms.

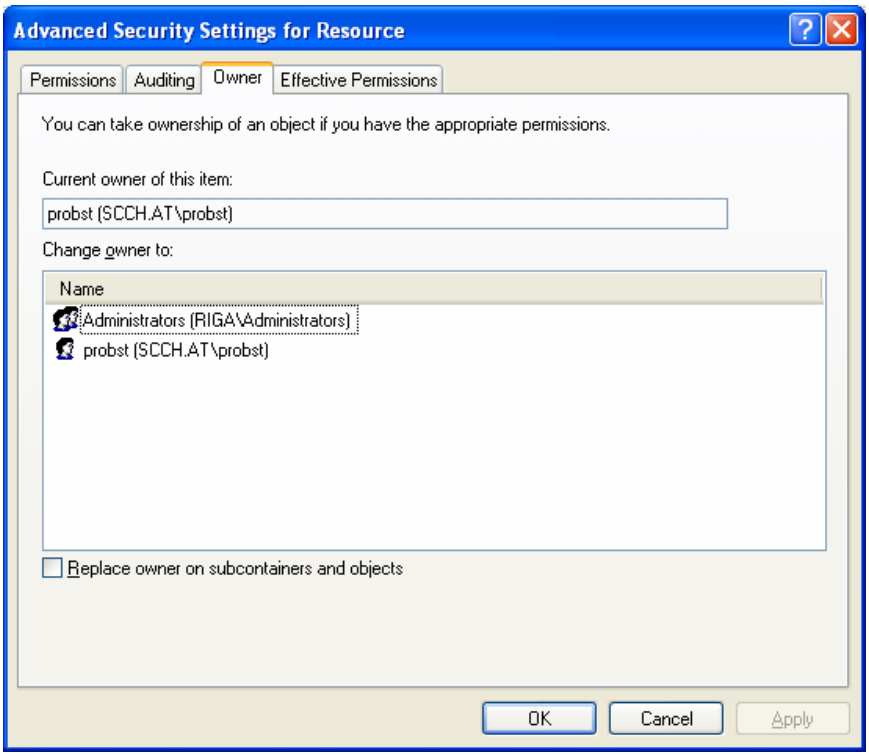


Figure 8: Ownership of a system resource in Windows

Nevertheless, authorizations can also be given to groups. Sandhu and Ahn (1998) state that groups can be seen as roles which leads to the assumption that Windows also provides RBAC features. Moreover, Windows 2000 supports role hierarchies because groups can be members of other groups. Windows only allows the activation of multiple groups at a single point in time since all groups a user belongs to are activated at login time. An explicit de-/activation of a single role is not supported.

The authorization model of Windows is centralized managed by having build-in administrator accounts. However, through membership of these administrator accounts, a decentralized management could also be implemented (*delegation*). Additionally the active directory can be scaled by separating the authorization base into subunits.

Summing up, one can say that Windows uses a combination of DAC and RBAC whereas the DAC model dominates over the RBAC mechanisms in order to meet the C2 security classification of the Department of Defense. Thus the owner of a

resource will always have the permission to modify access privileges although the role he is in denies this.

2.8.3 UNIX

Like Windows, UNIX heavily relies on the concept of owners. Each file in the system is assigned to an owner and a single user-group. The granularity of the UNIX authorization splits up into three categories:

- Permissions valid for the owner of the resource,
- The assigned user-group, and
- All the others.

Unlike in Windows, the granularity of UNIX is exclusive, which means that not the sum of all permissions is taken but the permissions defined for the user's category. For example, if read access is allowed to the group but not to the owner and the user is the owner, read access is denied although he is in the assigned user-group.

Ownership is assigned exclusively by the superuser (*root*). The owner itself is allowed to modify access privileges (read, write, and execute) onto his resource.

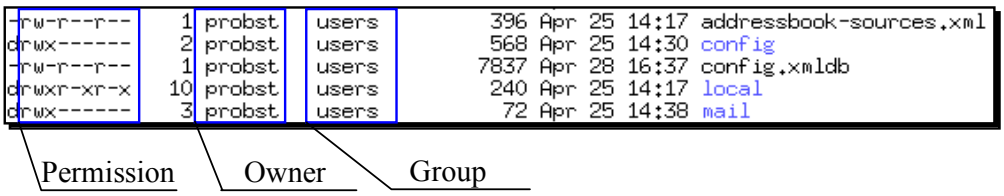


Figure 9: Permissions within the UNIX operating system

As already mentioned, UNIX supports the usage of groups which can be seen as roles (compare Sandhu and Ahn, 1998). UNIX itself does not support group hierarchies, thus only flat RBAC is realized in the system. The authorization model is strictly centralized since only a single user (*root*) is able to manage user and group accounts. However, the idea behind the RBAC model is that administration tasks are done by using an administrative role which is not supported in UNIX. Another issue is that UNIX supports only the membership of 16 or 32 (depending on the derivate) users per group. Sandhu and Ahn (1998) present a role-based extension to the system which supports a larger group-membership, group hierarchies and decentralized administration facilities.

Summing up one can say that UNIX heavily relies on the DAC approach whereas – when seeing groups as roles – only very few features of RBAC are supported.

The following contains a study of the integration of security mechanisms in relational database management systems. The RBAC aspects are taken from Ramaswamy and Sandhu (1998), whereas the study is extended by presenting other (e.g. DAC) approaches.

2.8.4 Informix DBMS

The Informix database provides a DAC model where the owner of a database resource is allowed to grant access to other users.

Informix also supports the RBAC model. A role can be granted to a single user, another role, a list of users, or to all users. A user can be granted to more than one role. Users who have been granted a role can further grant that role to another party or delete it. Informix has the restriction that a user can have only one role active in a single point in time. This implies that a user can act only in one role at any moment. Furthermore, there are no facilities to specify a default role to activate after login.

Informix provides features which enable nested roles (i.e. role hierarchies). However, there are no features to specify mutual exclusive roles, therefore no support for static separation of duties is available. There is no cardinality constraint to restrict the maximum or minimum number of users for a role. Dynamic separation of duties is not directly supported since multiple roles cannot be activated.

Privileges are subdivided into three different categories:

- Database-level privileges,
- Table-level privileges, and
- Execute privileges.

Database-level privileges refer to privileges needed to connect to a database, add new objects and perform administrative functions like security management or space management. Table-level privileges are needed for data transaction and querying. The execute privilege is needed to be able to execute stored procedures. The database administrator and the owner of a database object can grant privileges to a role and can revoke that privilege later on.

Summing up, Informix provides DAC and RBAC concepts. The RBAC itself supports role hierarchies but a user cannot act in more than one role at the same time. This can be quite a challenge in system administration since required permissions for different tasks must be assigned to single roles.

2.8.5 Sybase DBMS

Sybase also offers a combination of DAC and RBAC. Like Informix, an owner of a resource can grant access to his objects to other users. Furthermore, Sybase also allows granting access to a group of users.

In the case of RBAC, the Sybase DBMS comes with a set of predefined roles, the so-called system roles. These roles are the *sa-role* (System Administrator) for managing and maintaining all databases, the *sso-role* (System Security Officer) for performing security-related tasks and the *oper-role* (Operator) for backup and loading databases system-wide.

Within Sybase, a role can be granted to one or more users and any user can be granted more than one role, which is done by the System Security Officer. It is not possible for a user who has been granted a user-defined role to propagate that role to other users. This results in a stronger control over role assignments and proliferation. Sybase allows users to activate multiple roles in a user session. The activation process is required only for user-defined roles since system roles are automatically activated. It is possible to set up a default list of roles to be activated at the login time.

A role created can be granted to other roles and so a role hierarchy can be implemented. Sybase has the ability of defining two types of mutual role exclusion: *static exclusion* if a user cannot be granted both roles and *dynamic exclusion* if a user cannot activate or enable both roles at the same time. This feature allows static and dynamic separation of duties.

Sybase categorizes privileges in object access permissions and object creation permissions. Object access permissions are necessary to access data whereas object creation permissions grant the right to use commands that create objects. Both categories can be granted to roles. Since creation permissions cannot be granted with the *grant option*, the privileges to create new database objects cannot be propagated to other roles or users, thus only the System Security Officer can grant certain roles the permission to create new objects.

Again, one can clearly see that a combination of DAC and RBAC mechanisms is required. The RBAC model is quite mature since it supports role hierarchies and separation of duties.

2.8.6 Oracle Enterprise Server (DBMS)

The Oracle Enterprise server database management service supports both, DAC and RBAC mechanisms. Within the DAC model, the owner can give other users access to his resource using the `GRANT` statement. Furthermore, Oracle has a strong support of the RBAC model. Oracle supports role hierarchies and the delegation of rights by a special permission (*admin option*). Users who have been granted a role with this admin option can grant their roles to other users or roles. Additionally the user can steer the activation of roles at runtime since there are statements for the activation and deactivation of roles. Separation of duties is not supported.

Permissions can be defined on database objects or on the system whereas latter can be done only by users which have the admin option. Permissions on database objects can be assigned by the owner or by being member of a role that has the appropriate rights. System privileges are necessary to execute system commands. Object privileges allow users to perform actions on specific tables, views, sequences, or stored procedures.

Summing up one clearly sees that Oracle has a good integration of DAC and RBAC concepts whereas the RBAC implementation is very mature.

2.8.7 Summary

Summing up we can say that RBAC is not implemented very well in today's software products although the support in DBMSs is quite mature. We can think of various reasons for the lack of RBAC implementation. Windows for example aims to reach C2-level security and one of its most important requirements is that an owner of a resource must be able to control access to the resource – based on individuals and not on roles. Another reason is that design objectives want to reach other goals than RBAC; RBAC-related concepts have only been added later to those products.

In the case of operating systems, permissions can be granted to individual users as well as groups. Groups can be seen as roles but being able to assign permissions to individual users is not foreseen in RBAC. The activation of groups is done

automatically (all groups are activated at login time) but an explicit activation is not possible resulting in a lack of facilities for separation of duties.

One clearly sees that all presented products require a combination of DAC and RBAC, since RBAC alone would be too restrictive. Thus, this shows that the aimed solution needs to offer the possibility to combine several models to cover any security requirements. Furthermore, the aimed solution must provide a mature RBAC model, offering several aspects like separation of duties or role hierarchies, since RBAC is recognized as one of today's most important security models.

2.9 Support for Security Modeling through Development Platforms

Up to now, several security models and their presence in today's software applications have been presented. However, when talking about reusable security components, it is more interesting to know which support is provided by the development platforms. Thus, this subchapter concentrates on the support of developing security models using today's most important software development environments, namely the Java Platform (J2SE, J2EE) and Microsoft's .NET framework. Each platform and its offered mechanisms are presented first. Then a discussion follows showing the advantages and disadvantages. This will lead up to the motivation for GAMMA, a framework presented in Chapter 3, which offers reusable and transparent security components for application development.

2.9.1 Java 2 Standard Edition (J2SE)

Sun's Java 2 Platform (Sun, 1999) offers in its core a security model which only addresses the execution of code. This security model consists of the well known *sandbox model* and the so-called *protection domains*.

2.9.1.1 Sandbox Model

The sandbox model ensures that code is executed in a protected environment from which access to system resources like the file system or the network is very limited. This model is used when loading and executing applets over the Internet, where the code origin is unknown, thus the code is not trusted.

2.9.1.2 Protection Domains

The sandbox model is too restrictive for local applications, thus a second mechanism, namely the protection domains, is used to categorize code according to its origin. Privileges can then be assigned to these protection domains, allowing local code to access more resources (e.g. the local file system) and restrict mobile code obtained via the Internet.

2.9.1.3 Byte-Code Verifier

The byte code verifier screens the code to be sure that it was produced by a trustworthy compiler. This component checks the statements in the code if they are correct and do not produce an undefined or faulty state of the system.

2.9.1.4 Permissions

Permissions represent access to various system resources (e.g., files, network sockets) and are the core of Java's security system. Denials are not available, thus negative authorization does not exist within the Java security system. Furthermore, permissions are assigned to classes, meaning that a certain *Java class* is able to do something, not a specific user. Thus, the security system restricts Java classes, not users.

2.9.1.5 Policy

Mapping permissions to classes forms a policy, stating which classes may access which resources. A policy file is used to configure this policy for a particular implementation. This file can be manipulated either by using a text editor or by the graphical *policytool* which is part of the software development kit for Java.

2.9.1.6 Extensions

Java does not initially support security at a higher logical level. Thus the application developer has only limited means to integrate a security model. More complex mechanisms are provided as add-on libraries like the Java Authentication and Authorization Service (JAAS) that neatly integrates into the Java environment. Moreover, JAAS is part of the J2SE since JDK 1.4.

JAAS (JAAS, 2000; Lai et al., 1999) is designed to address multi-user environments by providing a framework and standard programming interface for authenticating users and assigning access privileges.

JAAS also addresses the problem that a subject may have multiple user-names. Nowadays, it is common that a user has different account names for various services. Thus, JAAS maps so-called *principals* which represents an account to a single subject. These principals have to be authenticated before the mapping to the subject is done. The authentication can be realized using multiple authorization modules. JAAS is therefore using plug-able modules (PAM, see Samar, 1996) which conform to the Generic Security Services Application Programmer's Interface (GSS-API, see Linn, 1997) and the Security Layer Application Programmer's Interface (SASL, see Myers, 1997) which provide a common base for various authentication systems. This enables single sign-on capabilities through the exchange of authentication tokens among various systems (e.g. the underlying operating system).

JAAS supports RBAC but in fact, it does not differ between roles, groups, and users. JAAS generalizes the different subject-classes to principals. Within the policy, access restrictions using permissions can only be made using these principals. Moreover, permissions only address limitations to use external resources. Restricting the execution of certain code parts has to be implemented by the programmer, e.g. code has to be added to check if the current user has a requested principal identity.

Another important part of a security system is the auditing component. Java provides a logging API (in `java.util.logging`) that offers logging and auditing mechanisms.

2.9.1.7 Discussion

Java is one of today's most common programming platform. Concerning security, Java concentrates on code security, meaning that Java protects its virtual machine against malicious code by providing a restricted environment (sandbox). The core Java security mechanisms do not address user-based security, thus it is not possible to formulate access restrictions on code pieces for certain users.

However, the Java Authorization and Authentication Service (JAAS) extends the core Java security by providing user-based security mechanisms. JAAS offers only programmatic security mechanisms, thus the application developer has to address security issues and has to make assumptions about the security requirements in the

target environment. Furthermore, if security requirements change over the lifetime of the software, the software has to be touched to modify the security relevant statements. This has negative impact on the maintainability, as well as on the reusability and flexibility of the resulting components and software pieces.

Another point is that JAAS only offers RBAC mechanisms whereas the realization is not conform to the NIST standard of RBAC. Other models can be realized using the JAAS mechanisms. This results in huge effort and restrictions as JAAS makes no distinction between various subject types. Thus a role, a group, or a user is the same for Java, since the security checks are done on the basis of principals and all subjects are mapped to such principals. This makes it tough to implement multiple security models (e.g., RBAC, DAC) and combine them, since a clear distinction between roles (used in the RBAC model) and users (used in the DAC model) is required.

However, the concept of principals as followed in Java has advantages. Principals allow more flexible models. For example, principals enable a role-based model where authorizations can be assigned to both, roles and users – thus a combination of DAC and RBAC characteristics. Having principals, an easy implementation of such a DAC-enhanced RBAC model can be provided.

2.9.2 Java 2 Enterprise Edition (J2EE)

The J2EE is the Java platform specially designed for enterprise servers (e.g., Bea WebLogic, IBM WebSphere). J2EE bases upon all mechanisms provided by the J2SE platform thus there is support for the sandbox model and the protection domains. However, J2EE provides additional support for security models which can be neatly integrated into applications (see Jendrock et al. 2002). For example, *Enterprise Java Beans* which are part of J2EE specify a role-based access control model which restricts the execution of methods on the basis of the role a user is acting in. Furthermore, this role definition and assignment is done declaratively which provides more flexibility and enables a later change in the security policy without having to modify the application.

2.9.2.1 Security Architecture

The J2EE security architecture consists of several layers, namely the *Transport level security*, the *Container security services* and the *EJB* or *web-tier Security model*. The Transport-layer security provides secure transfer of data over TCP/IP, using encryption techniques. The Container security services enforces the security policy

specified in the deployment descriptor of a web application. It is up to the container to protect the Java Virtual Machine and the host machine from unauthorized access by other applications. However, the J2EE standard does not prescribe how this protection has to be implemented, thus the security has to be provided with vendor specific solutions. The EJB or web-tier Security model is similar to the JAAS extension, offering roles to which principals are mapped and then evaluated by the container. J2EE Security Enforcement is done by the containers not by the components. This means that the container knows the security policy, the subjects and objects and monitors access to the methods of an object. The objects do not need to know anything about security because security is enforced outside the object.

2.9.2.2 Authorization (Access Control) Model

As mentioned above, J2EE uses declarative security, meaning that the code is protected externally, thus the code does not address access control itself. The big advantage of declarative security is enhanced reusability and flexibility, since only the externally stated security policy file has to be modified, if the security requirements change, the code itself does not need to be touched. Another advantage is that this model allows a clear separation of developer and domain expert tasks.

However, the declarative security mechanisms currently provided in J2EE are not expressiveness enough. For example, there is no possibility to use constraints for further restricting access (see Ziebermayr and Probst, 2004). Thus, it is only possible to grant a role the execution of a method or not. Often it is required to restrict access on the basis of the member of a role. In a banking application, the role *customer* is granted access on the method `ViewAccount(accountNo)`. It cannot be stated within J2EE, that a certain customer is only allowed to see *his* account, since J2EE evaluates access limitations on the basis of roles, not on individual users.

In order to address such access limitations, again programmatic security must be used, losing most of the advantages provided by declarative security mechanisms. Using programmatic security, the programmer has to fill his code with security regulation statements, which are very similar to the JAAS extension.

2.9.2.3 Authentication

Authentication within the J2EE environment is done by the container, thus the authentication scheme itself is container-specific. There are two possible authentication schemes. The first option is that the web-tier collects information

about a user and forwards them to the EJB-tier which then performs the authentication. The second possibility is that the web-tier collects information about a user and authenticates the user already in this stage. After authentication, the web-tier sends the identity of the user to the EJB-tier whereas the EJB-tier trusts the authentication performed at the web-tier.

The J2EE standard recommends several authentication mechanisms, but it is up to the vendor which mechanisms are supported. The recommended mechanisms are a simple username and password combination, X.509 certificates, or the integration into a Kerberos environment (the container must accept Kerberos tickets).

2.9.2.4 Discussion

The J2EE platform addresses mainly the usage of Java in the field of application servers. Thus, the J2EE extends the standard mechanisms of J2SE protecting not only the virtual machine but also the running applications within the J2EE environment.

J2EE provides declarative security mechanisms, allowing that the active security policy can be defined outside the application. This provides more flexibility, since the security policy can be changed anytime without having to touch the code. Furthermore, the code is not enriched with application-specific security statements, thus the code can be easily reused in other applications. However, the declarative security mechanisms provided in J2EE are not expressiveness enough to address more complex, security policies. In the case of a banking application, these declarative mechanisms would restrict the execution of a method depending on the role membership of an user (e.g. customer). This is not appropriate since the user should only be able to access his account. Thus, programmatic security is needed to be used to express those requirements. The programmatic security allows to enhance declarative security with constraints, resulting that J2EE offers better and more adequate means of implementing security than the J2SE platform. However, especially the mixture of programmatic and declarative security can become confusing, since the application developer has to address the constraints of a security policy by hand whereas the security policy itself is defined outside the application and can differ at the customer site from the developer’s assumptions. Moreover, the advantages of declarative security mechanisms are lost if the programmatic mechanisms are directly related to declarative ones.

Another point is that J2EE only provides a RBAC model out of the box. Other models can be implemented but only by using programmatic security mechanisms, losing again all advantages of declarative security.

An important issue when using J2EE security is the target environment of the J2EE platform. It is understood that mechanisms and security models implemented using the J2EE platform are only applicable in the field of application server, requiring a J2EE conform application server product. Thus, these mechanisms are not architecture and platform neutral and can only be used for a certain application domain.

Finally, J2EE only recommends certain mechanisms whereas it is up to the vendor of an application server which mechanisms are really supported (e.g. authentication mechanisms). This results in the fact that an application developer cannot be sure which mechanisms he can use. More important, applications can easily become dependent on a certain application server implementation.

2.9.3 Microsoft .NET

Another key-player in development platforms is the Microsoft .NET framework. This framework offers a wide variety of security features which enable the implementation and integration of high-level security models into applications. The following gives an overview on these mechanisms, describes their purposes and their usage. Detailed information can be found in Foundstone Inc. & CORE Security Technologies (2001).

2.9.3.1 Code Management

Within the .NET framework, code is executed in the *Common Language Runtime* (CLR). From the viewpoint of security, the CLR ensures that code can only act within a boundary defined through the security policy. Since all *managed code* is compiled and executed by the CLR, the CRL can react on unforeseen or malicious behavior of the code and take countermeasures. Thus, the CLR is responsible for access checking since all managed code has to pass it. However, .NET also offers the possibility of *unmanaged* code which bypasses the CLR and is executed directly on the machine. This code cannot be restricted by the CLR. Generally, unmanaged code should be avoided.

2.9.3.2 Evidence-based Security

Evidence-based security classifies code according to different criteria. It is up to the code to prove its trustworthiness by bringing an evidence. Depending on the conclusiveness of the code, it underlies certain user-defined restrictions. These restrictions are formulated in a *security policy*, which states what resources the code may access (e.g., usage of DNS-Server, file system, the system's registry), preventing software from errantly or malicious harming the integrity of data.

The policy is defined using *permissions*. Permissions describe a resource and the associated rights, and implement methods for demanding and asserting access. Thus there exist several permissions for different resources. Furthermore, the developer can extend the set of permissions to include application-defined resources and regulate access on them. This enables an adaptable security model to the aimed target-domain and can cover more complex security requirements of the application. Again, the CLR ensures that code is only loaded and executed if it has the sufficient permissions.

As mentioned above, the code must provide an evidence so that the CLR can assign the permissions defined in the security policy to the code. Evidence can come from several sources:

- Cryptographically sealed namespaces (strong names): Each class is part of a namespace. Namespaces can be sealed cryptographically by the user.
- Software publisher identity (Authenticode®)
- Code origin (URL, site, Internet Explorer Zone): The classification is based on the code origin. The security policy can classify code according to the URL or site it was loaded or according to the Internet Explorer Zone (Intranet, Internet, Local).

2.9.3.3 Code Access Security

The Code Access Security (CAS) is the enforcement engine that ensures that code does not exceed its granted permissions while it is executed. Code is permanently analyzed during runtime, to ensure that all operations are granted the needed permissions. Furthermore, CAS initiates a stack walk. This checks that each code in the call-chain has the demanded permission granted, not just the immediate caller. Stack walking prevents so-called luring attacks in which untrustworthy code attempts to trick code with greater access rights to call a protected object and bypass security restrictions.

2.9.3.4 Verification Process

The Verification Process addresses frequently but problematic errors like memory or buffer overflows or undefined process states. During the execution through the interpreter, the code is analyzed in order to detect such errors and immediately stopped if a certain operation would cause an undefined or faulty state.

2.9.3.5 Role-based Security

All points presented up to now are more security mechanisms than security models, but form a solid base for the development or integration of security models. It is up to the developer to build a security model using these mechanisms on the one hand, or to use the existing role-based access control model on the other hand. The RBAC model itself was already introduced earlier in this chapter. .NET provides *hierarchical RBAC* meaning that roles can contain a hierarchical structure. Nevertheless, *constrained RBAC* can be implemented with minor efforts by introducing verification processes for constraints.

The access verification procedure can be done in various ways, depending on the application domain. For standalone applications, the access verification procedure can rely on existing access control lists within the file system or is directly steered by calling the system function `IsUserInRole()`. This function is directly implemented in the code, protecting a code piece according to the caller's roles. The method itself determines the current user's roles and compare, if the user is a member of the required role. Depending on the membership, the function returns `true` (user is member) or `false` (user is not member). Normally, this function is integrated in an if-statement, thus the code-execution can be made dependent on the user's membership of certain roles. In the case of web applications, URL Authorization can be used, where access can be granted or revoked specifically by mapping users and roles to pieces of the URI namespace, including the request method (e.g., GET, HEAD, POST).

As mentioned in the initial discussion of security models, authentication is essential for a security model. Microsoft .NET provides various ways of authentication, realized through a set of authentication providers:

- *Form-based (Cookie) Authentication:* This provider invokes a specific HTML form on the client side. The user can then supply logon credentials and post them to the server which processes these token and performs the authentication. The credentials can be checked against different sources, such as a SQL database or a

MS-Exchange directory. In order to avoid multiple login dialogs during a session, the credentials are stored within a cookie, valid for a single user-session. The credentials are transmitted in clear-text, thus it is essential to use secure means of transmission (e.g. SSL).

- *Microsoft Passport Authentication:* This is a centralized authentication service provided by Microsoft offering single sign on capabilities for member-sites.
- *Internet Information Server Authentication Mechanisms:* The Internet Information Server provides several authentication mechanisms which can be used by .NET. These mechanisms include Basic Authentication, NTLM, Kerberos, Digest Authentication, and X.509 Certificates. For Basic Authentication and X.509 Certificates, the use of SSL is essential in order to provide a secure authentication. However, it is also advisable to use SSL for each authentication mechanism in order to avoid brute force attacks.
- *Windows Authentication:* The .NET framework can also rely on the Windows Authentication Mechanisms provided through the operating system. These mechanisms include Kerberos, NTLM, and X.509 Certificates.
- *Custom Authentication:* Developers can additionally write custom authentication and authorization code which can then be combined with the existing mechanism. Authentication providers can be configured per application and per virtual directory.

2.9.3.6 Other Security Mechanisms

The .NET framework additionally offers security mechanisms that are not part of the security model. They realize low-level security but can be used to realize a secure transmission of authentication tokens or to provide a solid base for a distributed security model. These mechanisms enable a direct integration into the Active Directory of a Windows 2000 domain, provide asymmetric (RSA, DAS) and symmetric (DES, 3-DES, RC2) encryption, hashing algorithms (MD5, SHA1, SHA-256, XMLDSIG), or supporting mechanisms for the Kerberos Protocol which enables a secure authentication.

2.9.3.7 Discussion

Currently the .NET framework becomes a big competitor to the Java platform. In comparison to the Java platform, .NET offers security on the basis of users and code, whereas the offered mechanisms are quite mature and very expressive. The .NET framework offers declarative and programmatic security mechanisms, whereas in the context of user-based security only the later is appropriate. Declarative security is

provided using customized attributes which can be defined in .NET. Customized attributes are placed within the code above the declaration of a method. Such attributes contain information that is evaluated by the just in time compiler. Currently these attributes allow the restriction of code, not the definition of access rules for users. However, it is imaginable to extend the .NET framework by a attribute-verificator that evaluates declarative access rules defined in the custom attributes. The programmatic security mechanisms are used by the integrated role-based access control model. Thus, the developer enriches his code with security-relevant statements, asking the runtime environment if the caller of the method is member of a required role. Depending on the role membership, code fragments are then executed or skipped. Like in Java, this programmatic security has negative impact on reusability and flexibility since the code must be touched every time the security policy is changing.

The .NET framework offers a role-based access control model. Other models can be implemented, with huge effort, since all programmatic mechanisms rely on the role-based model.

Another issue is the dependence on the .NET platform and the Microsoft environment. It is understood that the .NET environment can be easily and outstanding integrated into the Microsoft world. This results in the fact that better mechanisms can only be used wisely when relying on Microsoft products. Authentication is a good example. Simple authentication mechanisms (username and password) can be used with any product, better authentication mechanisms like NTLM or Windows Authentication can only be used in a Microsoft Windows environment. Furthermore, secure authentication and single sign-on using the Kerberos protocol can be done in any Kerberos environment. This integration is especially easy when using the Microsoft Kerberos implementation which is heavily used in Windows 2000 domains.

2.9.4 Summary

As shown in this subchapter, today's most used development platforms, namely Sun's Java and Microsoft's .NET supports developer in integrating security into their applications. However, both platforms have some weaknesses with negative impact on either the reusability or the flexibility of the resulting code. In the case of Java (J2SE), extensions (JAAS) are needed to offer programmatic security mechanisms. J2EE provides much better declarative security mechanisms, however they are not

expressive enough in order to address all security requirements. The .NET platform on the other hand comes with a rich set of mechanisms, however, again only programmatic security can be meaningful used in application development. Furthermore, security mechanisms offered are not platform independent, since they require either the Java or the .NET platform. In the case of J2EE there is an additional dependence on the used application server.

Thus, the aimed work tries to exceed the current limitations by providing reusable security mechanisms with *flexibility* as well as *platform and architecture independence*. This can only be archived by *declarative security mechanisms* that extend the used target environment.

Furthermore, today's programming environments only support the RBAC model. Thus, supporting *multiple security models* that are conform to existing standards and work *cooperatively* is another goal of this work. This raises the issue of needing a conflict solution mechanism if the models in use have different considerations about granting or denying access. The concept and realization of such a solution is shown beginning with the next chapter. However, before concentrating on our solution, an overview on related or similar work is given.

2.10 Related Work

The work described in this thesis tries to support software developers in integrating security into their applications. Thus, we looked on existing solutions for distributed authorization (compare Essmayr et al., 2001) and concentrated on solutions which can be used by software developers in order to enhance their applications with security mechanisms. Thus, the following will contain an overview on existing solutions.

2.10.1 RBAC Framework for Network Enterprises

Thomsen et al. (1998) define a framework for integrating RBAC into networked enterprises. Therefore they introduce seven abstract layers which are structured according to the responsibilities of the local system administrator and the application developer.

The first four layers deal with application specific constraints whereas the remaining layers focus on site-specific constraints. The intention of this approach is that application developers are creating the complex, application-specific security

constraints because application developers are the people who understand the application and its security requirements best. Administrators know the local security policy, thus they are able to manage the common site-specific constraints.

The layers themselves differ in the degree of granularity (e.g., objects, methods, a set of methods, application). An interesting fact is that authorizations can not only be assigned to an object and all its methods, but also to a group of methods. Another interesting feature is that the framework also provides a graphical policy setup tool, called NAPOLEON. This tool in combination with the layered framework allows that security policy managers can focus on the level of details they are familiar with, modifying details only as necessary.

2.10.2 RBAC Implementation Project (RIP)

Giuri (1998) shows a way to integrate the RBAC model into the standard Java platform (JDK 1.2). This is done by extending Java's standard mechanisms. The paper shows an implementation of a JDK extension in order to enable RBAC mechanisms within Java code, the so-called RBAC Implementation Project (RIP). RIP is an activity mainly devoted to the study and implementation of extension for available systems (e.g. Java) to provide affordable state-of-the-art role-based access control mechanisms.

The RBAC extension for Java follows the approach, that roles are derived from permission classes. RBAC itself is realized by providing a checking algorithm that controls and evaluates these role permissions. The extension allows a hierarchical role-structure, separation of duties, and constrained roles.

In comparison to JAAS (available for newer JDK versions), RIP offers on the one hand role-hierarchies and mechanisms for realizing separation of duties which is not supported by JAAS. JAAS, on the other hand, is not restricted to the role-based model but allows the realization of any security model, thus offering more flexibility and opens new ways for security and access control ideas.

2.10.3 Framework for Implementing RBAC using CORBA Security Services

Beznosov and Deng (1999) present a framework for implementing RBAC mechanisms using the CORBA security services. The authors use the security architecture of CORBA (the so-called CORBA security services) and extend them in order to use roles as subjects. CORBA itself sees users as principals, having a

component that authenticates principals (*PrincipalAuthenticator*) and another component that maps users to principals (*UserSponsor*). These components must be adapted, so that they can work with roles as well.

Within the paper, only flat RBAC and hierarchical RBAC are described. However, the authors show that other levels of RBAC can be implemented as well, requiring additional effort.

2.10.4 JSEF Framework

JSEF (Hauswirth et al., 2000) is a security framework that extends the Java 2 security architecture with higher-level security management and maintenance facilities especially for mobile code. The key features of JSEF are the hierarchical security policy supporting local, user-specific and global security policies, the system-wide security policy maintenance as well as concepts for user profiles and hierarchical user groups. The configuration of the security policy and the framework is done by graphical tools that generate XML documents.

The policy model of JSEF is derived from Java's standard policy model. This model provides enhanced policy semantics, a separation of local and global policy settings, and a dynamic policy negotiation component. JSEF supports positive and negative authorizations, called additive and subtractive permissions. The subtractive permissions always overrule the additive ones. These permissions and other JSEF mechanisms realize a RBAC-conform security model.

During runtime, a secure environment is established in which the mobile code is executed. As mentioned above, JSEF has different policies, namely the local, user-specific, and global security policy. The final policy valid for the secure environment is constructed by merging all these policies. A specialized security manager enforces the security by monitoring access to system resources and verify and check the properties of the mobile code during the runtime, avoiding malicious or non allowed statements.

As mentioned above, JSEF is concentrating on protecting mobile code. The framework just supports RBAC, other access control models or combinations of multiple security models are not supported. The interesting thing about JSEF is its ability to have a system-wide policy, able to merge global and local policy settings.

2.10.5 Kava Security Infrastructure

Kava (Welch and Stroud, 1999) is a security infrastructure that provides support for various security models and mechanisms in Java. The basic idea behind Kava is the use of a meta-object protocol that provides flexible and fine-grained control over the execution of components. This meta-object protocol maintains a meta-level security architecture that supports various security models.

Security is enforced through inserting security checks into the compiled code. This is done by using byte-code transformation which is steered by the meta-object protocol and the related meta-level security architecture. This results in the fact that Kava controls the invocation of methods, the object initialization and finalization, and state updates of the Java objects.

The approach used in Kava is the modification of the Java byte-code. Thus, the approach can only be applied to platforms, which use such an intermediate, interpretable language. Furthermore, this approach is not transparent to the developer since it is done in an extra step, outside of the developer's control. This can lead to unforeseeable behavior of the application, especially when relying on reusable components obtained from somewhere else. On the other side, Kava allows the realization and integration of various security models by providing an appropriate meta-level security architecture.

2.10.6 Other Related Work

The two systems presented below cannot be directly compared to the aimed GAMMA framework since they realize a distributed authorization solution instead of providing a supporting means for developers to integrate security models into applications. However, these systems were analyzed since important input can be gained from their architecture in providing modular security components.

SESAME (Ashley and Vandequauver, 1998; McMahon, 1994) provides an infrastructure for authentication, authorization and access control, as well as auditing. The system bases upon Kerberos and realizes a distributed security system. SESAME does not allow a customized security model nor the change of the underlying security model. Applications can use SESAME as a single-sign on facility, however it has not especially been developed for supporting application development by reusable components.

The *Adage* system (Zurko et al., 1999) provides authentication, authorization and access control, and auditing in distributed environments. Application developers can integrate this system and use its components via Adage's *Application Programming Interface* (API). Initially, RBAC is defined as Adage's security model but other models can be defined, requiring substantial effort to provide them. Nevertheless, Adage's architecture gives important input due to its modular structure that enables flexible and adaptable authorization solutions.

2.10.7 Discussion

A lot of work has already been done in order to provide security mechanisms that can be used in application development. However, each solution has its advantages and shortcomings which motivates the idea to develop a framework that actively supports application developers in integrating security models and mechanisms during the software creation process.

Such a framework has several requirements, which can be taken as criteria when looking and evaluating existing solutions.

First, the solution must actively *support the developer* during all stages of software development. Optimally, the application developer can choose which security model(s) he want to use and can define the security policy constantly during development. Furthermore, it should be possible to choose from different security policies in order to set up test cases for the software. During the early stages in the development process, there will be only assumptions concerning the security policy or the policy will not be addressed at all. Active support means that it should be possible to follow a security policy that allows all which seems to the application developer as if there is no security model active.

All these requirements can only be addressed by providing *declarative security* mechanisms. Thus, it is necessary to know, how security mechanisms can be *integrated* into the application (programmatic or declarative). Furthermore, it is interesting to know who is responsible for setting up the security policy.

Second, the solution should *not require special environments* which influence the aimed application domain. Thus, the solution should work with several platforms or provide its own environment that is applicable to various application domains.

An important point is the *expressiveness* of the security mechanisms. This means on the one hand that the security model must be aware and control all access to sensitive data, on the other hand the expressiveness of a security model states how mature the mechanisms are implemented and which features are available in order to address complex security requirements.

Another required feature is the support of *multiple models*. This requirements was already motivated when looking at today's software products and their security mechanisms. It was seen, that for example operating systems combine the ownership paradigm (DAC model) with the role-based model (RBAC), which provides much more flexibility and allows an easier administration. These criterions motivate the development of a new security framework called *GAMMA* (Generic Authorization Mechanisms for Multi-tier Applications) which is presented in detail in the following chapters. Chapter 5 then compares this framework with the aimed GAMMA security framework.

2.11 Summary

Within this chapter, security models, the idea behind them, and the need for them were presented. Furthermore, it was shown how security models are realized in today's software applications. Since this work concentrates on how security can be easily addressed in the software development process, the third part of this chapter showed the various ways how security mechanisms can be integrated into applications by software developers. This was done by analyzing how today's software development environments address this task by presenting some solutions that aims to facilitate such an integration.

The discussion showed that existing solutions still have some open issues when actively supporting application developers to realize security aware applications. Most solutions as well as today's software development environments, only support a certain security model whereas the usage of multiple models to cover more complex security requirements is hard or impossible to achieve. Another issue is that these solutions are often designed for a special platform or environment (e.g. Java) which often restricts the application domain. However, the most important point is the integration of these solutions into applications. Programmatic mechanisms can address complex requirements on the one hand but have negative impacts on reusability and flexibility on the other hand. Declarative security is often too restrictive or not expressive enough to address more complex requirements. Most of

the presented related work extends the standard Java security policy mechanisms in order to provide such a declarative security. This means that the user has to provide customized permissions that can be granted to a class or method which is rather a static assignment. Dynamic aspects such as activating or deactivating a role, separation of duties, or simple time constraints cannot be addressed using this policy facility.

Thus, the reader can easily see the need for a more intuitive solution that *supports the developer* in integrating security and provides an *architecture and platform neutral* set of reusable security components, mechanisms, and models that can be easily integrated into software applications. It is understood that these goals can only be achieved meaningful if the solution provides *declarative mechanisms* that allow flexibility and reusability on the one hand and that are expressive enough to cover also complex security requirements. Especially when trying to address complex security mechanisms, it must be a clear goal to support *multiple or customized security models*. Last but not least, the provided solution must be extremely *extendable* in order to provide a means that can be adapted to various application domains and future needs.

All these issues raised the idea to develop a new framework, called GAMMA, whereas the main motivation was to overcome these existing issues. The following chapter introduces the concept and design of GAMMA, its design goals and components. Chapter 4 then explains realization aspects and presents the JGAMMA reference implementation.

3 GAMMA

GAMMA (Generic Authorization Mechanisms for Multi-Tier Applications) is a security framework that helps to provide ready-to-use and adaptable security components and models.

This chapter introduces the GAMMA framework, describes its objectives, the architecture and components of the framework, and shows how security is enforced using the framework's mechanisms. This leads to Chapter 4 which presents the JGAMMA and GAMMA.net reference implementations that prove the framework's applicability. A short survey on GAMMA can be found in Probst et al. (2002).

3.1 Objective

The main objective of GAMMA is to provide ready-to-use security components that actively support application developers in integrating security mechanisms into their applications. Active support means that the application developer can rely on the security components already at the early stages of the development process. In fact, the framework must be able to provide a set of generic and common security components that can be transparently integrated into the application. As the application grows, it must be possible to adapt the security components without major effort to the evolving code. This is only possible by providing declarative security mechanisms which are steered outside the application by a security policy file. Decoupling the security layer from the application allows a maximum of flexibility on the one hand but allows on the other hand addressing complex security requirements of different application domains.

In order to make the framework usable, it must be able to address different application domains. Thus, the framework must provide generic mechanisms that do not require special architectures or platforms. As a result, the second objective of the GAMMA framework is to provide a platform and architecture neutral solution, meaning that GAMMA's concept can be realized in modern programming languages and environments. It is understood that reference implementations in certain programming languages will consider existing security mechanism and base on them,

as long as they are not in direct conflict with the platform neutral concept of GAMMA.

Another important point is that GAMMA must be highly extendable. This is necessary to reach the goal that GAMMA can be taken for any application domain, but also to allow the easy integration of new security models and mechanisms. Since security mechanisms will evolve in future, the integration of new mechanisms, ideas and concepts is very important to make the framework future-oriented and highly usable.

As mentioned in Chapter 2, a lot of today's software applications require a combination of two or more security models (e.g. RBAC and DAC combination). GAMMA must support such model combinations by providing ready-to-use models that can cooperate. Since conflicts are likely, the framework must provide adequate mechanisms to solve access decision conflicts of the various models in use.

3.2 Concept

This subchapter presents the architecture and design of the GAMMA framework, as well as the various mechanisms for providing security. Furthermore, it is shown how the objectives described in the previous subchapter – namely active development support, platform and architecture neutrality, expressive security mechanisms, the support of multiple models, and most importantly the support of declarative security – are reached by the concepts used in the framework.

3.2.1 System Context

The GAMMA framework provides the skeleton for security mechanisms such as authentication, authorization, and auditing.

The users of the framework are categorized into five groups as illustrated in Figure 10. This categorization is not intended to be exclusive, it is a categorization of interaction rather than of users and many users will belong to several groups.

- *Framework architects* are responsible for the design, implementation, maintenance as well as further development of the framework.
- *Model providers* are responsible for introducing new security models into the framework. The model can be introduced into the framework without modifying

the framework kernel. The framework kernel can only be modified by the framework architects.

- *Business application developers* are interested in developing software using the framework by integrating security mechanisms into their business applications. If they put anything into the framework, it is in terms of new ideas and visions that will be realized by framework developers.
- *Security administrators* are responsible for setting up the framework's mechanisms by administrating the single point of administration. Primarily, their work consists of realizing the security policy in defining the access rights and the mapping between the several domains and layers. In principle, they will work together with the business application developers.
- *Arbitrary users of business applications*: Any software written using GAMMA contains security mechanisms. The connection between users of these software products and the framework is the increased confidence in the software they are using.

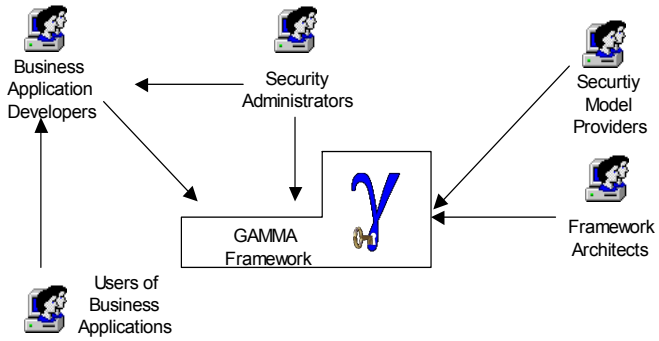


Figure 10: User groups for the GAMMA framework

3.2.2 Requirements to the Framework

In order to identify all requirements, several projects were analyzed and discussed. The analysis provided information about requirements, necessary issues, wishes, and nice-to-have features. The following contains a short description of the three systems that were analyzed.

3.2.2.1 Web-based Time Management

During a month, employees record their timetables. Each employee is the owner of his timetable, thus he has the full control over the document. The employee can pass

various rights to his document to third persons. This requirement can be best realized using the discretionary access control model.

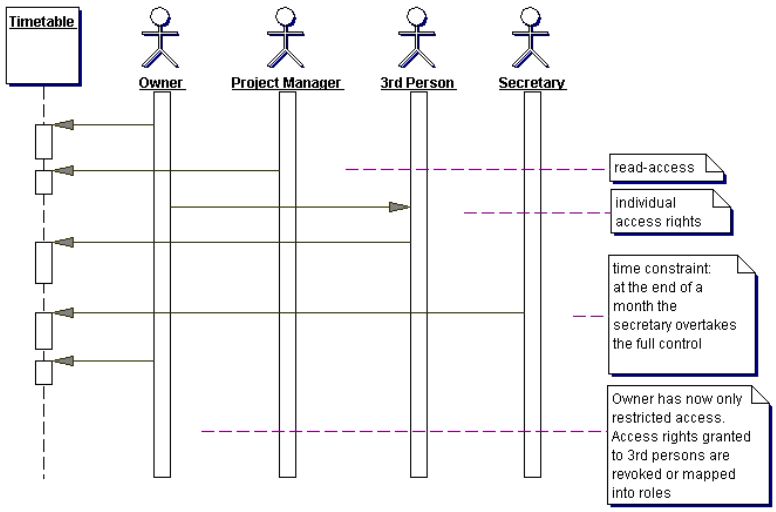


Figure 11: Sequence of web-based time management system

Additionally, there exists the restriction that the employee's project manager has always the right to read the timetable. This requirement can be best realized by additionally introducing the role-based access control model. The role "Project Manager" has always the right to read the timetables of his employees.

Thus, it is necessary that the security mechanism knows that the RBAC model is stronger than the DAC model. Although the employee has the full control over his document, he cannot deny the project manager's read access.

Furthermore, there is a big change in the access control schema, caused by a timely event. At the end of a month, the owner loses his full control over the timetable document. The dual access control model is changed into a single RBAC model. The role "secretary" now has full control over the document and is the only subject that is able to do modifications. The owner and the project manager have read-only access or when granted by the secretary also restricted write access. Subjects that had access to the object before – granted by the owner – now lose their access privileges.

3.2.2.2 FAW TISCover

TISCover is a web-based traveling guide system that allows an easy online reservation for hotels. Members can make offers by providing information within a TISPackage. First, the system administrator creates such a TISPackage. The system administrator has full rights over this package, thus he is able to grant / revoke rights

on the package to other subadmins. A subadmin then sells a package to a TISUser who becomes the owner of this package. Per default – as long as there is no other specification – the owner has full rights over the package, except the right to grant access to other users. If other users have to access the package, the subadmin has to grant access. This access is only content-based, thus only content of the package can be changed. A user then interacts with the content in the package and gets results of the interaction. If the subadmin has the right "helpdesk", he is able to take over the function of the owner or other TISUsers who have access to the package in order to perform some administration tasks on the content of the package.

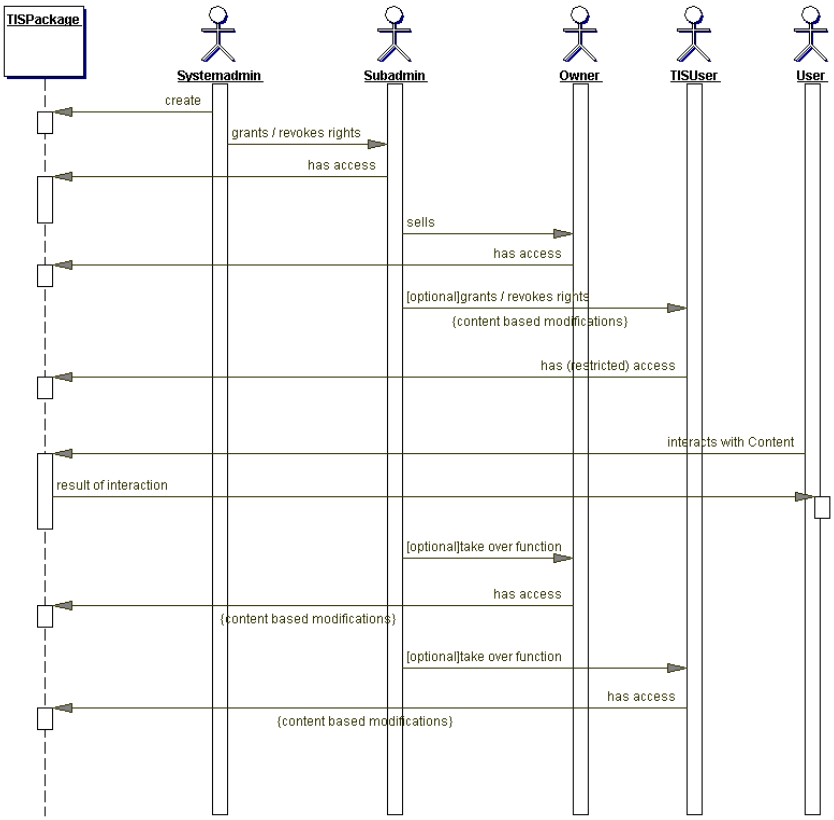


Figure 12: Sequence of TISCover System

In general, the assignment of rights can be separated into the following subjects:

- the *system administrator* has the rights to administer the system and adjust or modify the system environment,
- the *subadmin*, who is able to modify the system environment for the areas he is assigned to, and
- the *TISPackage* that contains rights based on its content.

On top of the subject-hierarchy there is the TISUser from whom all other roles are derived. The assignment of rights is done on the level of TISPackages, which means that it is defined who is able to access which content of the package.

Rights are hierarchically inherited. A member of a higher hierarchy-level has automatically all rights of the members of a lower level. However, there are rights that can be inherited and rights that cannot. This requires additional flexibility of the permission hierarchy.

Objects within packages inherit the rights of the package. An explicit assignment of permissions is not provided.

3.2.2.3 SCCH Intranet Project

The SCCH Intranet project aims to provide a platform for corporate intranets. The case study concentrated on the planned security layer of this Intranet project. The general concept is described in Figure 13.

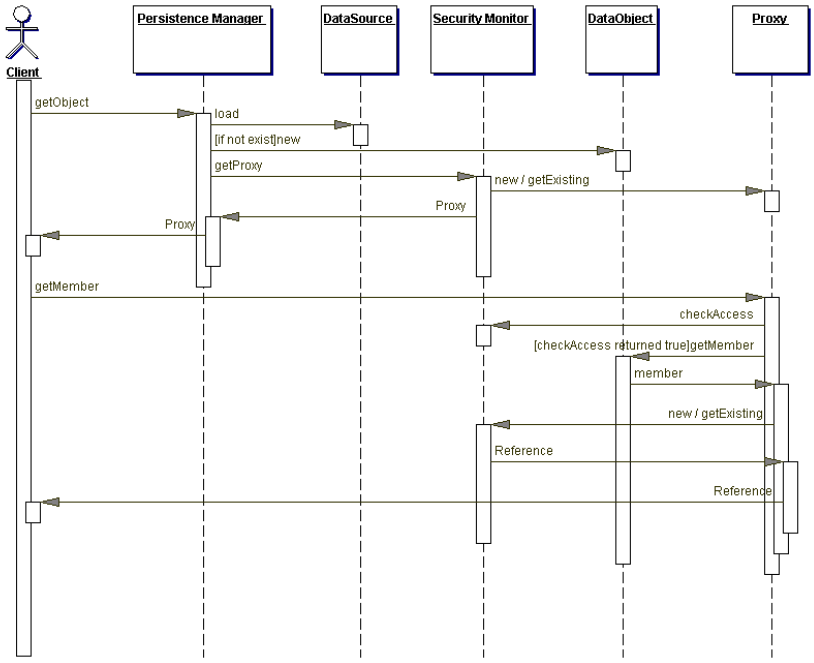


Figure 13: Sequence of proxy concept

The client or subject wants access to a data object. This object is stored persistently and must be retrieved via a data source first. Thus the client invokes the `getObject()` method on the persistence manager. The persistence manager contacts in a first step the data source and delegates the load request. If the data object is not yet loaded, the persistence manager creates a new instance of the data

object. Instead of returning a direct reference to the data object, the persistence manager contacts the security monitor and requests a proxy for the data object. The security monitor checks if there exists a proxy for the requesting subject. If so, it returns a reference to the proxy; if not, the security monitor generates a new proxy and returns it to the persistence manager. The persistence manager returns the retrieved proxy reference to the client.

If the client wants to invoke a method or retrieve a member variable of the data object, it has to pass the proxy-class to the data object are made through the proxy. The proxy invokes an access control check on the security monitor. Only if the check succeeds, the proxy retrieves the requested member from the data object. Since the retrieved member can be also a complex object, it is sometimes necessary to contact the security monitor which generates a proxy for the retrieved member object. The proxy itself returns a proxy to the requested member object. Once again, all access is done via proxies.

Within the system, there are various levels of permissions. Levels are depending on the granularity of the security object.

- *Permission at attribute level*: These permissions define, if a subject can read or write an attribute of an object. If the subject has insufficient permissions, an exception is thrown.
- *Permission at instance level*: These permissions define, if a subject can access a reference or instance of an object. In order to be able to gain a reference, the subject must have at least the read permission to an object, otherwise the security monitor's `getProxy()` method returns `null`. Without the write permission, the subject is not able to write attributes of the object, except this is explicitly granted at the attribute level.
- *Permission at class level*: These permissions define, if a subject can access objects of a certain type. A subject which is denied access at this level cannot access any concrete instance of the class as long as there are no explicit permissions at instance level.
- *Permissions at class and attribute level*: These permissions define the default behavior when access requests to attributes of objects are made. These permissions can be overwritten using permissions at object or attribute level.

3.2.2.4 Identified Security Requirements

During the studies, a lot of security requirements were identified. Since some cases require specialized security components, the requirements were generalized and are described in the following.

Combinable security models: Sometimes it is necessary that one or more models are combined and work together at the same time. In this case, it must be possible to specify, which model is dominating in which respect. Since each model has evaluation rules which are looked up when verifying a request, the domination of the model defines the order of the rules lookup. Each model must be expressed in rules containing constraints and model-specific data, whereas special rules regulate the coexistence between the models. The access controller is delegating access requests to the models in the specified order. The models evaluate the request according to the rules. This evaluation depends on the assumption of the security model. There are two possible assumptions: the open and closed world assumption. Within the open world assumption all accesses are granted except those which are explicitly denied by rules. This requires prohibitions (negative authorization). The closed world assumption denies all requests except those which are explicitly granted by rules. According to the evaluation result of the rules, the model returns a positive answer (a rule was found which grants the request), a negative answer (a rule was found which denies the request) or no answer (no rule was found). If the model returns a positive or negative answer, the request is granted or permitted. If the model returns no answer, the request is sent to the next model. If all models return no answer, the request is evaluated according to the world assumption.

Constraints: Constraints realize restrictions within a security model. Constraints could be categorized into two groups: model dependent constraints that are only applicable to a certain security model and model independent constraints that can be used with various security models. However, model independent constraints must react on events which are raised by specific security models. In the following there are some examples for such constraints. This list is understood as a basic set of constraints. The framework itself should provide an abstract base class for a later implementation of additional constraints.

- *Model dependent constraints*: These constraints depend on a specific security model. Since the role-based access control model (RBAC) is especially addressed in this work due to its importance, only RBAC-related constraints are mentioned. The most important constraint within the RBAC model is the so-called separation

of duties. Depending on the time this constraint is active (runtime or administration-time), one distinguishes between static and dynamic separation of duties. Both constraints deal with conflicts of interests in a role-based system. *Static separation of duties* (SSD) prevents conflicts of interests by constraining the assignment of users to roles. This means that if a user is authorized as a member of one role, the user cannot be a member of a conflicting second role. This constraint is defined already before the administrator assigns users to roles, preventing the administrator to assign a user to conflicting roles. *Dynamic separation of duties* (DSD) deals with the issue of conflicting roles at runtime. It is possible that two roles are not directly in a conflicting state but only if a member activates these two roles at the same time, the conflict arises. However, it is allowed that the user may activate the second role if he deactivates the first role before. Since the conflict arises at runtime, no objections are made if the administrator is assigning the user to both roles.

- *Model independent constraints*: These constraints are independent from the current active model. Nonetheless, they are in a certain relationship with the security models since these have to interpret the constraints correctly and determine the effects on the model. Model independent constraints should be encapsulated from the security models and provide a generic interface which is then used by the current active security model. Examples of such constraints are the *time constraint* restricting access depending on the current time, *location constraint* restricting access depending on the location of the subject, *history constraint* restricting access on the subject's history of actions.

Inheritance and granularity of permissions: Containers, like the TISPackage, are special components containing a set of other objects. Permissions are assigned only to the container. Objects within the container inherit automatically the permissions assigned to the container they reside in. However, it is sometimes necessary to override this permission inheritance and provide means for defining a new permission assignment for sub-containers or items.

The concept is similar to the Windows NT file permission assignment. Permissions assigned to a folder are automatically reflected to all items within this folder (subfolders and files). However, it is possible to assign other permissions to files or to subfolders. In the latter case, items in the subfolders reflect the permissions assigned to the subfolder.

Security Model Requirements: Since security models form the central part in access control, it is important to identify requirements to these models. Each model comes with its specific requirements which cannot be identified completely in advance. It is the model provider's task to identify these requirements and develop the security model following GAMMA design patterns. The following illustrates this issue according to two examples.

Within the discretionary access control (DAC) model, it might be necessary that the ownership privilege is circulating between various users within the system. This means that the administrator of a system can revoke the ownership privilege from a user and grant this privilege to another user. This process also includes a variety of dangers and requires special treatments. Methods and ways must be defined when the ownership privilege is removed and assigned to another user. Since the owner is allowed to grant individual rights to other users, these permissions must also be treated (e.g. revoked). In fact, there exist several possibilities how to treat the revocation of ownership (cascading deletion, deletion without cascading, deletion without effects). It is up to the model provider which possibilities are supported by his model.

In the case of RBAC, the model can be expressed as a structure of graphs (e.g., authorization, role-hierarchy). The graph realizes the inheritance relation of each hierarchy. Since multiple-inheritance should be supported – at least within the role hierarchy – a simple tree structure is not sufficient. However, it should also be possible to model the hierarchy using a tree or other graph structure. Thus, this model requires a flexible structure of graphs.

Restricted inheritance of permissions: When building up the permission hierarchy it is sometimes necessary to restrict the inheritance of permission, especially when supporting multiple inheritance. When building up the permission hierarchy, means to define the restriction of inheritance within the role- and permission hierarchy are necessary.

Restricting result sets: Access control does not only mean to restrict the calling of specific methods, also the returning result must be controlled. Depending on the permissions a request may return different results. This can be realized either by filtering data already at the data producer's side (e.g. on the server) or by the local access control system (e.g. remove non-allowed data before forwarded to the application). For the sake of performance and transport security, the first solution

should be aimed. However, it is not always possible to influence the data producer, thus sometimes the second solution is required.

Transparent security mechanisms: Security mechanisms should be transparent and neatly integrated into the development of the application. In the optimal case, the developer does not need to take special care of the security mechanisms. This means that it should not be necessary to write special code in order to integrate security mechanisms. The security mechanisms should be present in an encapsulated layer which can be changed at any time. It is rather unrealistic that the final security policy is already available at development time. Thus the security mechanisms should support a modifiable policy, enabling security administrators at the target's site to define the security policy according to their needs. The security policy is defined in external files and described by all the parts of the security description language (SDL).

However, it is impossible to hide security completely from the developer. The aim is to relieve the developer from the burden to implement security components and to make the integration as easy as possible. This can be reached by offering various security components and objects from which the developer can derive and generate secured objects. The framework must then take care that these secured objects are integrated into and protected by the GAMMA framework.

Flexible security mechanisms: Since security requirements can change over the lifetime of an application, the mechanisms must be adaptable. For the sake of maintenance and reuse, the optimal case contains declarative security mechanisms.

3.2.3 Architecture

This subchapter deals with the overall architecture of the GAMMA framework. First it motivates the major design criteria of the framework. Furthermore, it contains the component diagram listing the various framework components, showing their interactions, relationships, navigability, and usage.

3.2.3.1 Major Design Criteria

The following lists the major design criteria that must be considered to meet the requirements stated above.

Clear separation between "data models": In order to introduce new security models, separating the underlying data models is necessary. Each security model has

at least a subject-, an object, and an authorization model. A constraint model can be added too. These models have to be clearly separated so that each data model can be changed or extended. The concrete security model realizes the interaction between these data models.

Common base for all security models: To allow an easy extension of the framework, each security model must have a common base. Our framework does this by transforming each model into a rule base. The rule base has a standardized layout and contains access rule entries. If a model needs a completely new rule base, it can extend the standard one and provide its own rule base. Invoking the check methods of the entries (e.g., permission, constraint) performs the rule checking.

Encapsulated "extension code" localized in defined places: One of the design goals is to provide a highly extensible framework. However, extensions of the framework must be coordinated. Therefore various roles for extending the framework were introduced. Each role can only extend the framework in defined places. The framework architect has a specific knowledge of the framework and can extend the infrastructure and framework components. On the other side, the model provider does not have specific knowledge about the structure of the framework, thus the role is only allowed to introduce new models and model-related components.

Well defined "interfaces" and as "generic" as possible: Each component of the architecture will implement a number of interfaces used for interacting with the other components. Each interface consists of a set of functions which are specialized for some type of interactions. The intention is to define these interfaces in a way as generic as possible. That is, they should be independent of the actual component's implementation and also of the concrete data types that will be added by the users when customizing the framework.

Separation between "persistent data" and "transient data": In order to remain independent from the physical storage of data, the persistent data must be separated from the transient representation. In fact, for each transient storage there would be a component which is aware of obtaining the data from various persistent storages. Thus a base class is provided containing the transient storage. Implementations will deal with obtaining the data from various kinds of physical storages (e.g., XML-file, database, operating system).

Re-use standard components: The intention is to provide a generic framework that is neatly integrated into the used programming platform. The framework itself is

understood as a concept that enables the easy integration of security components into applications. Reference implementations, written in a specific programming language, prove the applicability of the framework. It is foreseeable that these programming languages already offer some security components. The intention is to analyze these features and integrate them as good as possible into the reference implementation. Furthermore, the framework must base upon existing technology to provide state-of-the art work. This means that security models have to be implemented according to standards or to use open and widely accepted formats (e.g. XML).

3.2.3.2 Security Definition Language

The security definition language (SDL), based on the XML standard, is responsible for the configuration of the framework. It deals with the models which have to be initialized as well as with the security policy. Furthermore, it states which data provider is used for gathering the framework's required information.

As already mentioned, the models which act within the GAMMA framework are the essential part of the SDL. The order in which the different models appear within the SDL file defines the policy. Furthermore the world-assumption is assigned to each model (closed world or open world; see Chapter 2.2.4).

The values which are provided by the configuration file represent classes which are loaded by the framework itself. Using this mechanism, it is guaranteed that the framework acts as generic as possible, a separate class can be taken for each data provider.

3.2.3.3 Component Diagram

Before the components are described in detail, the main components of the framework are shown using a component diagram (Figure 14). The diagram illustrates the decomposition of the system.

The various framework components are described in detail in the next subchapter. They are not meant to be complete and in their final form. In fact, the shown components are the basis for the framework.

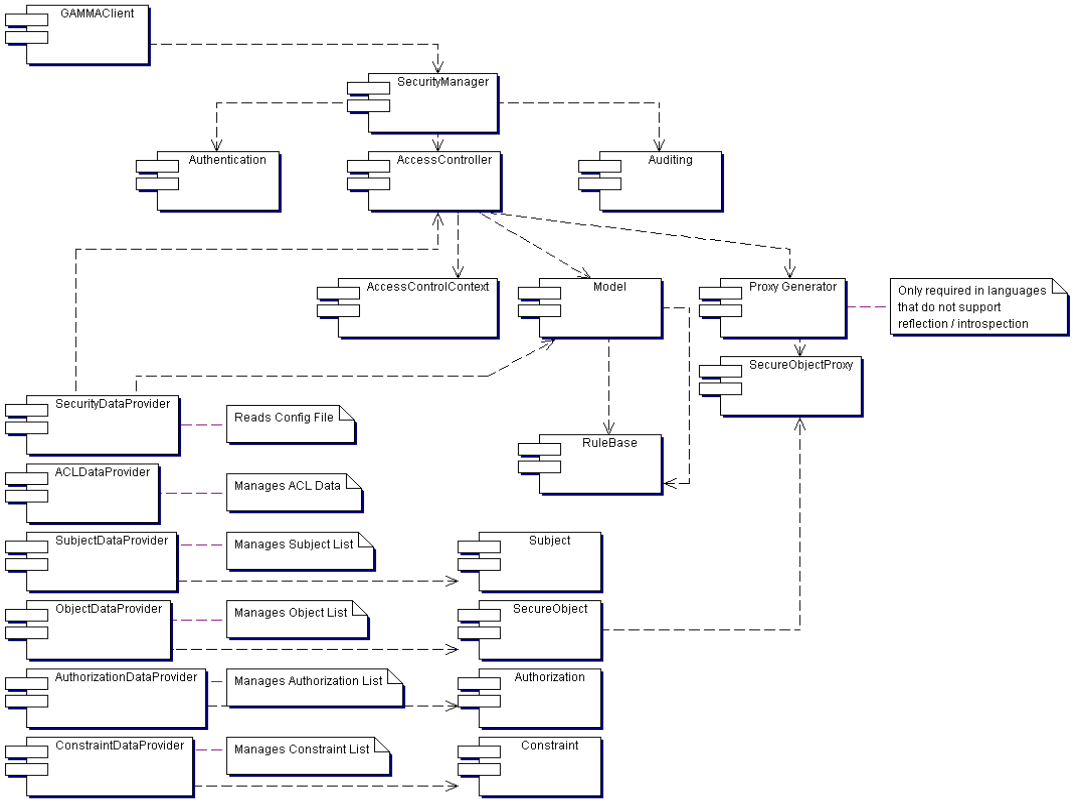


Figure 14: Component Diagram of GAMMA

3.2.3.4 Classification of Components

In the following the main components which will form the kernel of the architecture are shown. These components are categorized according to their supplier role on the one hand, and according to their functionality on the other hand. Some components are thus listed twice because various users will extend different functionality within the components.

Table 1 shows the components which are visible for and thus extendable by the application developers. Developers are able to integrate new data providers which cover storage formats that are used at the customer's site. Furthermore, the Secure Object Data Provider is able to manipulate the various instances of secure objects which implement the business logic.

Table 2 lists the components which are maintained by the framework architects. These components mainly reside in the framework's kernel. An extension requires in-depth knowledge of the framework and its mechanisms.

Subject Data Provider	Retrieves subjects from a persistent storage and transforms them into a transient representation. Furthermore, it creates subject instances. For each type of storage the framework user can implement his own Subject Data Provider. Usually there is one Subject Data Provider per model.
Secure Object Data Provider	Retrieves secure objects from a persistent storage and transforms them into a transient representation. Furthermore, it creates secure object instances. For each type of storage and for each concrete implementation of secure objects the framework user can implement his own Secure Object Data Provider. Since an application can have various secure objects, several Secure Object Data Providers can be used by a single application.
Authorization Data Provider	Retrieves authorizations from a persistent storage and transforms them into a transient representation. Furthermore, it creates authorization instances. For each type of storage the framework user can implement his own Authorization Data Provider. Usually there is one Authorization Data Provider per application.
Constraint Data Provider	Retrieves constraints from a persistent storage and transforms them into a transient representation. Furthermore, it creates constraint instances. For each type of storage the framework user can implement his own Constraint Data Provider. Usually there is one Constraint Data Provider per application.
ACL Data Provider	Retrieves ACL entries from a persistent storage and transforms them into a transient representation. Usually there is one ACL Data Provider per model.

Table 1: Components supplied for the application developers

Security Manager	One per application. Manages the whole application and connects the three tasks *authorization*, *authentication*, and *auditing*.
Security Data Provider	One per application. Retrieves the security policy, expressed in the configuration file and sets up the framework and security models using the SDL and data providers.
ACL Manipulator	One per Rule Base / Model. Creates and manages the rules for the Rule Base.
Subject	Realizes actors and entities of the system.
Secure Object	Provides a base class for all objects which shall be protected.
Authorization	Realizes a concrete access right onto a resource.
Constraint	Restricts actions within the system or enables additional restrictions in access rules.
Access Controller	One per application: Receives access requests and dispatches them to the current active models and controls the combination of security models.
Access Control Context	One per request: Realizes a transient store for access pattern tuples and further meta information.
Rule Base	One per Model: Realizes a transient storage containing access rules valid for a security model.
Constraint Verificator	One per Model: Realizes a transient store containing constraints valid for a certain rule within the Rule Base.
Proxy Generator	One per application: Is aware of generating Secure Object Wrappers (Proxies) for Secure Objects.
Secure Object Wrapper	Realizes a proxy to a Secure Object.

Table 2: Components supplied for the framework architects

Model providers mainly extend the framework by introducing new security models and their semantics. Table 3 shows the components which are mainly used to realize security models.

Model	Provides an abstract base class for a concrete security model. In principal, it realizes a real world security model.
Subject Data Provider	Normally one per Model: Transfers subjects out of a persistent storage into the model. Furthermore, it prepares subject data for models (e.g. users are assigned to roles and roles are forwarded to the model).
Object Data Provider	Normally one per Model: is the connection between the application and the security model. The application developer can request protection for an object using the Object Data Provider. Furthermore, it prepares objects for models (e.g. assigning a security level when using MAC).
Authorization Data Provider	Only for special use: Transfers authorization data into the model and prepares them for the model. A specialized Authorization Data Provider can prepare certain authorizations for concrete models.
Constraint Data Provider	Only for special use: Transfers constraints into the model and prepares them for the model. A specialized Constraint Data Provider can translate model-independent constraints into model-dependent ones.
ACL Data Provider	Normally one per Model: Transfers ACL entries into the model and prepares them for the model. Special models need special ACL entries which vary from the defined structure. These entries can be realized using a model dependent ACL Data Provider.

Table 3: Components supplied for the Model Provider

3.2.4 Security Mechanisms

This subsection presents an overview on the security mechanisms offered by GAMMA but also shows how these mechanisms work together.

In principle, GAMMA offers components for authentication, authorization and access control, and auditing which are described in Chapter 3.2.5. During the interaction with the framework, the user must be first authenticated. This is necessary since all other components rely on this authentication. Both, authorization and auditing components must know the user that performs certain actions.

The user's identity is first proofed by the authentication component. In order to use the framework, the user must provide a valid identity (e.g. username) and a corresponding identifier (e.g. password). The framework evaluates these tokens within the authentication component. GAMMA allows various authentication methods due to its open architecture and the support of the GSS-API (see Linn, 1997).

Access control is done in various steps: In general, a subject wants to access an object in a certain way. The requested operation on the object defines the necessary authorization that is needed in order to fulfill the task.

Primarily, the security manager – which represents the interface to the client – receives a request from an authenticated subject for a certain operation on an object. This request is delegated to the access controller that passes the request to all active models – in the order that is specified in the security policy. Each model searches for a subject/object pattern in the rule base, which matches the request. The search process returns a list of possible authorizations that are defined for the subject/object combination. Each authorization is explicitly checked by invoking the authorization's `checkAccess()` method. Depending on the result and the world assumption of the model, the authorization check is either positive (access is granted) or negative (access is denied). However, it is possible to define additional constraints that further restrict a specified access operation to an object by a subject. Thus, each related constraint is evaluated. Again, the decision is delegated by invoking the `checkAccess()` method of the constraint object. If both, the authorization and the constraint grant access, access to the object is permitted. This result is returned to the access controller and then returned to the security manager. The mechanism is illustrated in Figure 15.

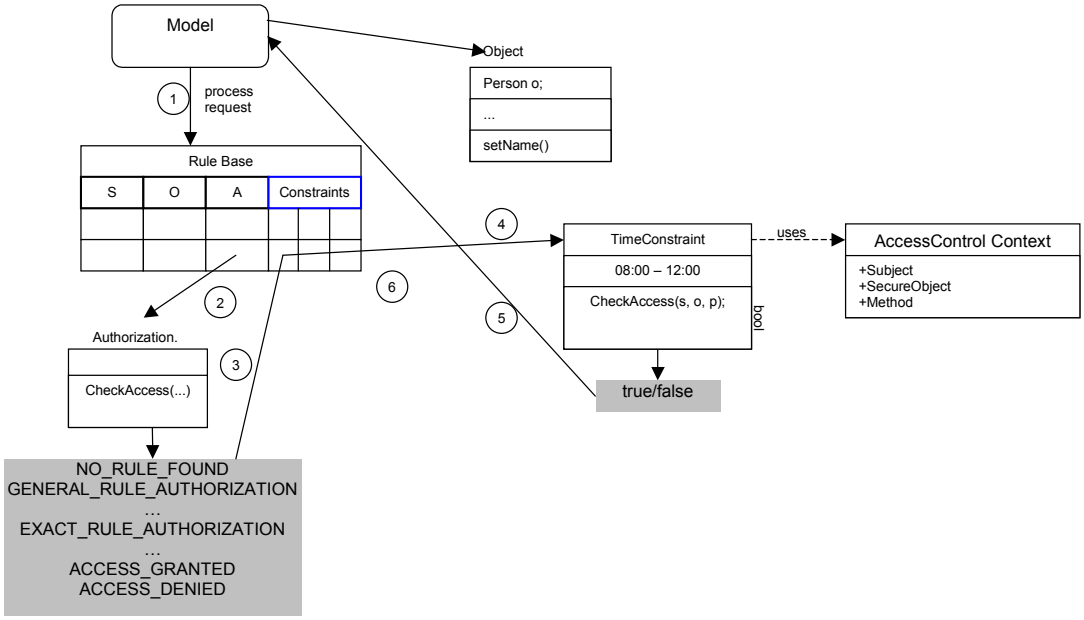


Figure 15: Access Checking mechanism

3.2.5 Components

This subchapter discusses the various components of the framework in more detail. Since a lot of abstract base classes are provided, which must be inherited for realizing concrete tasks, these are described as well as final components which perform various tasks. The usage of the components will be explicitly mentioned.

3.2.5.1 Security Data Provider

The Security Data Provider is responsible for managing the setup of the framework and takes care that the security policy defined in the Security Definition Language (SDL) is transferred from its persistent representation into the transient storages of the framework.

The main task of the Security Data Provider is to read the configuration file which is created by the security administrator. According to the entries of the configuration file, the Security Data Provider retrieves the actual security policy and the combination and relationship of models. Furthermore, the configuration file contains references to the data provider for each model which are responsible for retrieving the data. Thus the Security Data Provider is responsible for setting up and controlling the model's data provider.

The Security Data Provider interacts with the ACL Data Provider from which it gets ACL Tuples. The Security Data Provider fills the tuples with object references to

subjects, secure objects, authorizations, and constraints. These objects are created by the corresponding data providers. If a tuple cannot be resolved due to missing entities within the data store, the tuple is removed and ignored. However, it is imaginable that an entry in the audit trail is created to document this incident. In a last step the Security Data Provider delivers the filled tuple to the appropriate model which in fact transfers the tuple into the Rule Base using the ACL Manipulator.

Since the flow of SDL data is bi-directional, the Security Data Provider is also aware of retrieving the tuples from the model respectively from the Rule Base and split them into their parts. Each part is handed over to the appropriate provider which is aware of writing the data back into the persistent storage. Additionally, since the Security Data Provider is responsible for the setup of the framework, it offers a means for storing modifications of the configuration back to the configuration file. Thus, it has to interact with the access controller for transferring the current security policy.

Dependencies: The Security Data Provider depends on the configuration file and the data providers.

Context: The Security Data Provider interacts with the Access Controller, the models, the configuration file and the various data providers.

Interfaces: The Security Data Provider provides the following interfaces:

- `ISecurityDataProvider` defines the functionality of the security data provider, thus it contains all methods necessary for manipulating the security policy (SDL).
- `IFrameworkConfiguration` defines methods necessary for handling the framework's configuration file and the setup of the framework's components (especially models and data providers).

3.2.5.2 ACL Data Provider

The ACL Data Provider retrieves ACL tuples in the form of subject, object, authorization, and constraint from the persistent ACL storage (SDL). The tuple contains references to the requested objects that are resolved by the Security Data Provider and replaced by object references retrieved from the data providers. Only secure objects are replaced by placeholders (containing the hash code) and can be referenced only at runtime.

When writing the tuples to the persistent storage, the ACL Data Provider assigns object references within the tuple. The objects are stored using the data providers.

In fact, the ACL Data Provider is an abstract base class for retrieving ACL data. Concrete implementations cover different storage types.

Dependencies: The ACL Data Provider depends on the ACL storage and the Security Data Provider.

Context: The ACL Data Provider interacts with the Security Data Provider, providing the ACL entries.

Interfaces: The ACL Data Provider provides the following interfaces:

- `IACLDataProvider` defines the functionality of the ACL Data Provider such as managing and manipulating ACL tuples.
- `IStorage` defines storage capabilities.

3.2.5.3 Subject Data Provider

The Subject Data Provider realizes an abstract base class for retrieving subject data. Concrete implementations will deal with two main topics: retrieval from different storage types and retrieval of different subject types. In fact, security models have their own requirements to subjects. An RBAC model needs roles as subjects whereas a DAC needs users. Each concrete data provider has to manipulate its data so that it fits into the model. For example the role data provider has to provide roles as subjects but is also responsible for the role user mapping. This is done in the way that the subject data provider obtains a reference to the model and therefore can interact with the model. On the other hand, subject data can be stored in various physical data storages (e.g., database, operating system, LDAP) thus different providers are needed which are aware of handling such storages.

The Subject Data Provider retrieves subject data from a storage. Then it creates an instance of a subject class, feeding this instance with all data retrieved from the storage. The object reference is given to the Security Data Provider and stored in the Security Data Provider's tuple.

Dependencies: The Subject Data Provider depends on the subject storage, on the Subject component, on the Security Data Provider, and on the Model.

Context: The Subject Data Provider interacts with the Security Data Provider, providing the subject data and the Model in order to perform a model specific user mapping.

Interfaces: The Subject Data Provider provides the following interfaces:

- `ISubjectDataProvider` defines the functionality of the Subject Data Provider such as managing subjects out of a storage.
- `ISubjectQuery` defines query possibilities for subjects used by the various concrete security models.
- `IStorage` defines storage capabilities.

3.2.5.4 Object Data Provider

The word *object* is used in the following in the sense of objects containing sensible data that needs to be protected by the security system. The Object Data Provider offers an interface for transferring an object's hash code to the Rule Base. Whenever an application programmer creates a new object and wants to protect it, the object is given to the Object Data Provider which reads out the hash code and provides it to the Security Data Provider.

Tuples (Subject, Secure Object, Permission, Constraints) do not contain object references themselves but a unique id, identifying the object. The Object Data Provider is only used when creating new objects during run-time.

Dependencies: The Object Data Provider depends on the object storage, the Secure Object Component, and on the Security Data Provider.

Context: The Object Data Provider interacts with the Security Data Provider, providing the object data.

Interfaces: The Object Data Provider provides the following interfaces:

- `IObjectDataProvider` defines the functionality of the Object Data Provider such as managing and registering objects to the security system.
- `IStorage` defines storage capabilities.

3.2.5.5 Authorization Data Provider

The Authorization Data Provider realizes an abstract base class for retrieving authorizations. Authorizations can be either positive (permissions) or negative (prohibitions). Concrete implementations will cover various storage types. The Authorization Data Provider transforms transient authorization data out of and into a persistent storage. When retrieving the data out of a storage it creates an instance of the authorization object and initializes the object. The authorization object reference is then given to the Security Data Provider and stored in the Security Data Provider's tuple.

Dependencies: The Authorization Data Provider depends on the authorization storage, the concrete Authorization component, and on the Security Data Provider.

Context: The Authorization Data Provider interacts with the Security Data Provider, providing the authorization data.

Interfaces: The Authorization Data Provider provides the following interfaces:

- `IAuthorizationDataProvider` defines the functionality of the Authorization Data Provider such as handling and managing authorizations obtained from a storage.
- `IStorage` defines storage capabilities.

3.2.5.6 Constraint Data Provider

The Constraint Data Provider realizes an abstract base class for retrieving constraints. There are various kinds of constraints which will be mentioned later in this chapter. Concrete implementations will cover various storage types. The Constraint Data Provider can retrieve constraint data from or store it into a persistent storage. It is aware of creating instances of the constraint objects using the right subclass and initializes the object. The type is received from the storage. The constraint object reference is then given to the Security Data Provider and stored in its tuple. Figure 16 shows an example representation of persistent constraint data in a database. The Constraint Data Provider knows which concrete constraint class has to be used by interpreting the type. The Constraint data is stored as meta data within the database. This data is interpreted by the constraint component which queries through the meta data.

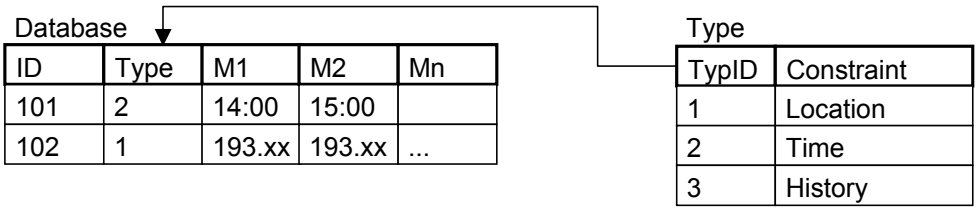


Figure 16: Example representation of persistent constraint data

Dependencies: The Constraint Data Provider depends on the constraint storage, the concrete Constraint component, and on the Security Data Provider.

Context: The Constraint Data Provider interacts with the Security Data Provider, providing the constraint data.

Interfaces: The Constraint Data Provider provides the following interfaces:

- `IConstraintDataProvider` defines the functionality of the Constraint Data Provider such as handling and managing constraints obtained from a storage.
- `IStorage` defines storage capabilities.

3.2.5.7 User Data Provider

The User Data Provider (UDP) is a special subject data provider, providing subjects for a model. Since each model needs users, this is the only realized subcomponent of the subject data provider which is part of the framework kernel. The UDP obtains user data from a persistent storage and provides these data within a transient storage. For each type of persistent storage a concrete implementation of the UDP must be provided. Since the transient storage is independent from the persistent storage and looks equal for each storage type, the transient storage is implemented in the base class.

Dependencies: The UDP depends on the Subject Data Provider.

Context: The UDP works together with the Subject Data Provider, thus also with the Security Data Provider which is responsible for writing the subject references to the ACL tuples.

Interfaces: Since the UDP is a special subject data provider, it provides the same interfaces.

3.2.5.8 Model

The Model component is the abstract base class for a concrete security model. It collects subjects, objects, authorizations, and constraints from the Security Data Provider and transfers them to the Rule Base respectively the Constraint Verificator using the ACL Manipulator. The model has to be aware of the underlying assumption to perform the access control. When the model is delegated to check access requests, it generates a search pattern and executes the pattern matching process in the Rule Base. Then the result is evaluated according to the world assumption and one of the following results is returned:

- *True*: when a rule was found and access is granted
- *False*: when a rule was found and access is denied
- *Weak True*: when no rule was found in an open world assumption
- *Weak False*: when no rule was found in a closed world assumption

Furthermore, the model is responsible for model specific tasks (e.g., role activation, owner grants permissions to another subject).

Dependencies: The Model depends on the Access controller, the Subject, Object, Permission, and Constraint component.

Context: The Model interacts with the Rule Base, the Constraint Verificator, and the ACL Manipulator.

Interfaces: The Model provides the following interfaces:

- `IModel` defines the functionality of the Model component such as the management of security models. Furthermore, it contains the access check functionality for a security model. The Access Controller will delegate the access check via this interface.
- `ISecurityDataManipulaton` defines the functionality of obtaining security data from the Security Data Provider. The security data is then forwarded to the ACL Manipulator (`IRuleManagement` interface).

3.2.5.9 Security Manager

The Security Manager is the central manager of the framework. It controls all security components. Requests are posted to the security manager, whereas the security manager dispatches them to the corresponding components. Moreover, it handles the interaction of the authentication, access control, and auditing part. The

security manager has a bootstrapping method which initializes the standard components of the framework.

Dependencies: None.

Context: The Security Manager controls the whole framework. Thus it interacts with all parts and components of it. Furthermore, it is the interface to the framework's clients.

Interfaces: The Security Manager provides the following interface:

- `ISecurityManager` defines the functionality of the Security Manager. In fact, the framework user (the program using the framework) will interact with the GAMMA framework only via this interface.

3.2.5.10 ACL Manipulator

The ACL Manipulator creates and manages the rules for the Rule Base. Thus, it manages the transient storage of the Rule Base. In principal, the Rule Base has to be changed according to the following incidents:

- Change request from an authorized subject (e.g. owner grants rights to another person). The subject may differ depending on the current model in use.
- Creation of new objects which have to be protected.

The ACL Manipulator creates an entry in the Rule Base (rule) with the information obtained from the Model respective the Security Data Provider.

For generating the rule, the component uses subject, object, and authorization information. For each entry in the Rule Base it assigns an identification number which provides the connection to the list stored in the Constraint Verificator. Within the same step the entries (tuples) in the Constraint Verificator are created containing the identification number of the Rule Base and the constraint objects.

Typical tasks of the ACL manipulator are:

- Add a rule to the Rule Base
- Modify a rule within the Rule Base
- Remove a rule from the Rule Base

Performing tasks on a rule implies the modification of the corresponding entries in the Constraint Verificator.

Dependencies: The Model uses the ACL Manipulator for generating the Rule Base.

Context: The ACL Manipulator interacts with the Constraints Verificator, the Security Data Provider, Subject, Object, Authorization, Constraint, Rule Base, and the Model Component.

Interfaces: The ACL Manipulator provides the following interfaces:

- `IACLManipulator` defines the functionality of the ACL Manipulator. The model component is communicating with the ACL Manipulator via this interface.
- `IRuleManagement` defines the functionality for creating and managing rules which are then given to the Rule Base (via the `IChangeRuleBase` interface).

3.2.5.11 Subject

This component realizes actors and entities of the system, such as persons, processes, or model specific entities (e.g. roles).

Sometimes a subject executes a method on an object which requires access to another object. In order to perform this task, there are two possibilities:

1.) The object becomes a subject and the access control is done by verifying if the corresponding object has the right to access the other object.
2.) The first object is calling the method of the second object on behalf of the subject. Therefore, the subject needs sufficient access rights to the second object. This method is called transitive access.

In the case of accessing an object transitively, the subject of the first object is taken for access control decision (compare to Figure 17). This subject, as a part of the request parameters, is stored in the Access Control Context. When access to the second object is requested, the access controller gets the subject from the Access Control Context and checks whether or not the subject is allowed to call the method on the second object.

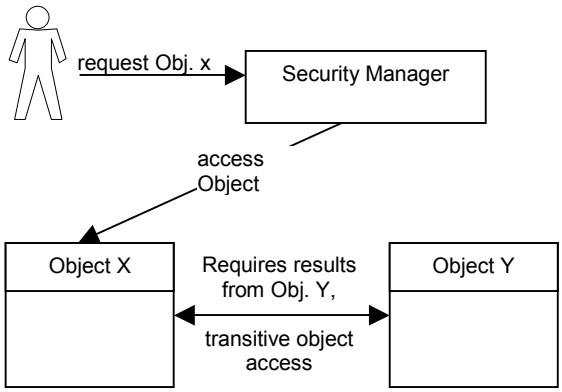


Figure 17: Transitive Object Access

Subjects are both, model specific and general (e.g. users). An abstract base class is provided for implementing new types of subjects (compare to Figure 18).

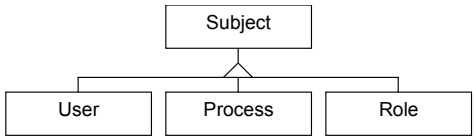


Figure 18: Abstract base class and concrete subclasses for subjects

Dependencies: None.

Context: Since subjects are one of the most important components, they interact with nearly every other component. In fact, they interact with the Security Manager, the ACL Manipulator, the Access Controller, the Constraint Verificator, the Subject Data Provider, the Model and its Rule Base, and the Access Control Context.

Interfaces: The Subject component provides the following interface:

- `ISubject` defines the functionality of the subject component.

3.2.5.12 Secure Object

The Secure Object is the base class for all objects to be protected by the GAMMA system. It is necessary that a client does not obtain a direct reference to the object. By declaring the constructor *private*, only a privileged component can create the Secure Object. This raises the need for a factory (see Gamma et al, 1995), which constructs the secure object and returns an appropriate wrapper. In fact, the Proxy Generator is responsible for the task of creating the object or obtaining a reference if the object already exists.

Dependencies: The Secure Object depends on the Proxy Generator and the Secure Object Wrapper.

Context: The secure object interacts with the Proxy Generator, the Secure Object Wrapper, the Object Data Provider, the Model, and the Access Controller.

Interfaces: The Secure Object component provides the following interface:

- `ISecureObject` defines the functionality of the secure object component.

3.2.5.13 Authorization

The Authorization Component realizes access rights onto resources. Authorizations are logically separated from the security model. This separation enables the use of the same authorization component in various security models. Authorizations are based upon Secure Objects not upon subjects. Thus, authorizations are assigned to objects, stating the allowed actions on the object that can be done by certain subjects.

Authorizations are implemented and have a defined meaning for a given object. To separate the authorization logic from the object model, each authorization component must implement the authorization semantic itself. Thus, the component provides a `checkAccess()` method. This method is called when access needs to be validated. In fact, the authorization component decides itself whether access can be granted or not.

Authorizations can be either positive or negative. Positive authorizations are called permissions whereas negative authorizations are called prohibitions. Authorizations are used within the Rule Base. The Rule Base contains tuple entries containing a Subject, Object, and Authorization. If a subject requests access, it generates a pattern that is compared to the entries in the Rule Base. If a fitting entry is found, the Rule Base delegates the evaluation to the Authorization object that offers this `checkAccess()` method. The method interprets subject and object according to the authorization model. In fact, it uses meta information for verifying the possibility of accessing the resource in the requested way. Dependent on the type of Authorization, the object tells the Rule Base if access is granted or denied. In order to perform its work, the component resolves the granularity of the object that is divided into the following levels:

- *Class*: Permissions are defined for classes, so each instance (object) inherits the permissions defined on the class

- *Object*: Permissions are defined for a concrete object instance.
- *Method*: Access to a method of a class or object is requested.
- *Member*: Access to a member variable is requested.

When the granularity is resolved, the check method analyzes the protected resource and determines if access is possible. If, for example, read access is requested and granted by the security policy on object level but the object does not provide any means for obtaining information (e.g. get-methods), access cannot be granted and has to be denied. The check method returns true if access can be granted, otherwise false.

However, the authorization component is an abstract base class for various access rights. The framework will provide some basis authorizations which can be either positive (permission) or negative (prohibition) depending on the model's world assumption (e.g., read, write, execute for the object access; read, write, modify, execute for file access). Such authorizations are called *assumption-based* authorizations. The presence of an authorization means that the rule falsifies the model's world assumption, saying that a privilege should allow something in a closed world or the very same privilege should deny something in an open world.

By extending the Authorization component, individual authorizations can be created.

Dependencies: The Authorization component depends on the Authorization Data Provider which provides the authorization data and creates the object.

Context: The Authorization component interacts with the Security Manager, the ACL Manipulator, the Access Controller, the Rule Base, the Access Control Context, and the Authorization Data Provider.

Interfaces: The Authorization component provides the following interfaces:

- `IAuthorization` defines the functionality of the authorization component.
- `ICheckRule` is used to delegate the access check evaluation. This interface defines the functionality to evaluate access requests according to the authorization.

3.2.5.14 Permission

A Permission component explicitly grants access to a resource. Thus, the permission does not base on the model's assumption. If the Rule Base finds a rule which permits

access, the access is granted to the corresponding subject. If no rule is found, the model relies on its world assumption.

Dependencies: The Permission component depends on the Authorization component.

Context: The Permission component interacts with the Security Manager, the ACL Manipulator, the Access Controller, the Rule Base, the Access Control Context, and the Authorization Data Provider.

Interfaces: The Permission component is a specialized authorization thus it provides the same interfaces.

3.2.5.15 Prohibitions

Prohibitions explicitly deny access to a resource. Like the Permission component, prohibitions do not base on the model's assumption. If the rule base finds a rule which prohibits access, the access is denied to the subject. If no rule is found, the model relies on its world assumption.

Dependencies: The Prohibition component depends on the Authorization component.

Context: The Prohibition component interacts with the Security Manager, the ACL Manipulator, the Access controller, the Rule Base, the Access Control Context, and the Authorization Data Provider.

Interfaces: The Prohibition component is a specialized authorization thus it provides the same interfaces.

3.2.5.16 Constraints

Constraints restrict certain actions within the system. Constraints define conditions which allow or deny the execution of certain tasks, thus there are positive and negative constraints. Furthermore one differentiates *model specific* constraints and *model unspecific* constraints.

Model specific constraints influence only actions and tasks of the security model and are defined and implemented by the model provider. Examples for such constraints are separation of duties, respective static separation of duties and dynamic separation of duties. Model unspecific constraints are independent from the current active

security model and influence the application as a whole. These constraints are defined and implemented by the framework architect. The independent constraints are further classified according to their function. Examples are location-based constraints (e.g. logins are only allowed from specific IP addressed) and time based constraints (e.g., access to resource is not allowed from 8pm to 6am, access is only allowed over a duration of 2 hours after login).

Constraints are verified by the Constraint Verificator which checks the constraint condition after successful verification through the access controller. Constraint information are provided through the access control context which contains information needed by the constraint to verify the access request.

The Constraint Verificator delegates the verification task to the constraint which provides a check method. This method returns true if the current rule is valid and thus should be considered.

Dependencies: None.

Context: The Constraint component interacts with the Rule Base, the Model, the Security Data Provider, the ACL Manipulator, and the Constraint Data Provider. Furthermore, for gathering information about the access request, the Constraint component has to interact with the Access Control Context. Since constraints may depend on permissions, the constraints also interact with the Authorization component.

Interfaces: The Constraint component provides the following interfaces:

- `IConstraint` defines the functionality of the constraint component.
- `ICheckRule` is used to delegate the access check evaluation. This interface defines the functionality to evaluate access requests according to the constraint.

3.2.5.17 Access Controller

The Access Controller receives access requests and dispatches them to the current active models. The request parameters are stored in the Access Control Context which is considered when access control decisions are made. Furthermore the Access Controller is responsible for delegating the authorization checks to the models corresponding to the policy stated in the SDL. The Access Controller dispatches the requests to the active models in the order specified in the SDL and awaits the result of the rule evaluation. A model can return one of the four values “access granted”,

“access denied”, “weak access granted”, or “weak access denied”. If a strong result is returned the access controller stops the request handling and returns the result to the security manager. If a weak result is returned, the next active model is contacted. The first weak result is valid in the case of there is no strong result returned by any model.

Finally, the access controller is responsible for the security layer which contains the various proxy classes. Thus, it controls and invokes the Proxy Generator which generates Secure Object Wrappers. The Access Controller returns these wrapper classes to the requesting subject.

Dependencies: The Access Controller depends on the Security Manager and the Model for handling access requests and the Proxy Generator for generating secure objects.

Context: The Access Controller interacts with the Security Manager which manages the overall security within the system, the Model to which it delegates access requests, the Access Control Context storing information for further access requests and constraint evaluations, and Subjects, Objects, and Authorizations.

Interfaces: The Access Controller provides the following interfaces:

- `IAccessController` defines the functionality of the Access Controller. The Security Manager interacts with the Access Controller via this interface.
- `IACCManagement` provides an interface to the Access Control Context.

3.2.5.18 Access Control Context

The Access Control Context realizes a transient store for access pattern tuples (subject, object, authorization) and further meta information (e.g., target, host, time). The Access Control Pattern provides information for requesting meta information for constraints or authorizations. The meta information is extensible and can contain further entries in the context. This information is stored in a sort of list and the corresponding constraint or authorization objects can retrieve the information via access methods.

Dependencies: The Access Control Context depends on the Model, Subjects, Objects, and Authorizations.

Context: The Access Control Context interacts with Permissions, Subjects, and Objects which are stored in the access control context store. Furthermore, the store can contain framework independent data within its list. The Rule Base respective Authorizations and Constraints components can obtain this information for their check methods.

Interfaces: The Access Control Context provides the following interface:

- `IAccessControlContext` defines the functionality of the Access Control Context. The Access Controller interacts with the Access Control Context via this interface.

3.2.5.19 Rule Base

The Rule Base is a transient storage including tuples in the form of an id, a Subject, an Object, a Class, or a Method, and a certain Authorization. The id of each tuple is used for referencing entries within a Constraint Verificator object. The constraint reference is not mandatory, as not each subject or object has constraints.

Generally, the Rule Base is a transient store that contains an integrator which accepts a pattern containing the search condition for a rule. Furthermore, the Rule Base is aware of adding and removing model based rule tuples. The Rule Base itself is managed by its corresponding Model component. In fact, every Model has its own Rule Base containing the Model's specific data.

Every model has to break down its structures to the Rule Base. In detail this means that for every model subjects, authorizations, and constraints have to be implemented which are stored within the Rule Base.

Dependencies: The Rule Base depends on the Model component since every Model has its own Rule Base. Furthermore, the `ACLManipulator` component is responsible for the dynamic behavior of the Rule Base.

Context: The Rule Base interacts with the Model, Subjects, Secure Objects, Authorizations, and the Constraint Verificator.

Interfaces: The Rule Base provides the following interfaces:

- `IRuleBase` defines the functionality of the Rule Base. The ACL Manipulator interacts with the Rule Base via this interface.

- `IQueryRuleBase` provides querying functionality within the Rule Base. The Model performs its pattern matching process via this interface.
- `IStorage` defines storage capabilities.

3.2.5.20 Constraint Verificator

The Constraint Verificator component owns a transient storage including tuples consisting of an id and a Constraint. The Verificatior proves the validity of a Constraint by calling the Constraint's check method.

A further task of the Constraint Verificator is the management of the transient storage which holds the id-Constraint tuples.

Dependencies: The Constraint Verificator depends on the Rule Base and the Model.

Context: The Constraint Verificator interacts with the Rule Base, Constraints, Authorizations, Subjects, and Secure Objects. Furthermore, the ACL Manipulator is responsible for managing the changes within the Rule Base – respectively the Constraint Verificator. The Constraint Data Provider maps the persistent representation of the constraints to transient ones. The Access Control Context, as the storage for a request, provides further information for the built-in check method.

Interfaces: The Constraint Verificator provides the following interfaces:

- `IConstraintVerificator` defines the functionality of the Constraint Verificator.
- `ICheckConstraints` is used for the interaction with the Constraint Verificator. This interface provides querying functionality within the constraint storage. The Model and the Rule Base perform their pattern matching process via this interface.

3.2.5.21 Proxy Generator

On request of the Access Controller, the Proxy Generator generates a Secure Object Wrapper. If the object to protect already exists, a reference to it is obtained, otherwise the object is created. Using runtime meta-information, a proxy (Secure Object Wrapper) is created which contains the method stubs of the protected object. These stubs perform security checks and invoke the object's corresponding method if the check is positive.

Dependencies: The Proxy Generator depends on the Secure Object and the Secure Object Wrapper.

Interfaces: The Proxy Generator provides the following interface:

- `IProxyGenerator` defines the functionality of the Proxy Generator. The Access Controller uses this interface to perform the proxy generation task.

3.2.5.22 Secure Object Wrapper

The Secure Object Wrapper realizes a proxy to a Secure Object. It contains stubs to the Secure Object's methods. The structure of the method stubs can be described as following: First, a security check is done which examines whether the subject is allowed to call the Secure Object's method. The Access Controller is responsible for determining the access control conditions. If the check is positive, the real method of the Secure Object is invoked in a second step (compare Figure 19).

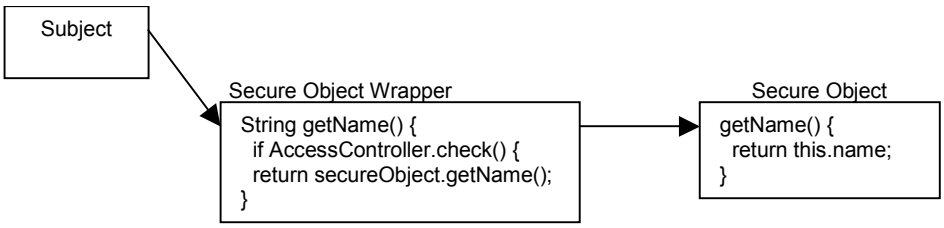


Figure 19: Secure Object Wrapper

Dependencies: The Secure Object Wrapper depends on the Secure Object, the Access Controller, and the Proxy Generator.

Context: The Secure Object Wrapper interacts with the Secure Object, the Access Controller, and the Proxy Generator.

Interfaces: The Secure Object Wrapper provides the following interfaces:

- `ISecureObject` defines the functionality of the Secure Object Wrapper. Since each Secure Object Wrapper is a proxy to a Secure Object, the Secure Object and the Secure Object Wrapper implements the same interfaces.

3.2.6 What GAMMA Can Do

GAMMA is a generic, adaptable, extensible, and modular framework for integrating security in business applications. GAMMA allows the efficient re-use of existing

security mechanisms in any kind of business applications. Moreover, GAMMA can be adapted to specific application domains and thus supports application developers in increasing their software quality.

In general, GAMMA can be used for any kind of applications. However, certain types of applications need a specific adoption of the framework which can be done by extending the framework's components.

GAMMA is typically used in server environments. Objects residing at the server are protected by the GAMMA environment and can be accessed by clients using the framework client (`GAMMAClient` component). However, since GAMMA concentrates on high-level security it takes security at a lower level for granted. If an underlying component is not secure (e.g., operating system, database), GAMMA will not provide adequate security.

3.2.7 Integration of GAMMA into Applications

GAMMA integrates by introducing a new layer to a multi-tier architecture. This new layer is placed between the data and the business logic that processes the data. Thus the business logic is only allowed to access the data if the security system renders the access positive. By defining the access privileges via a declarative language, the application does not need to know anything about security restrictions or requirements. The security policy is enforced by the security layer.

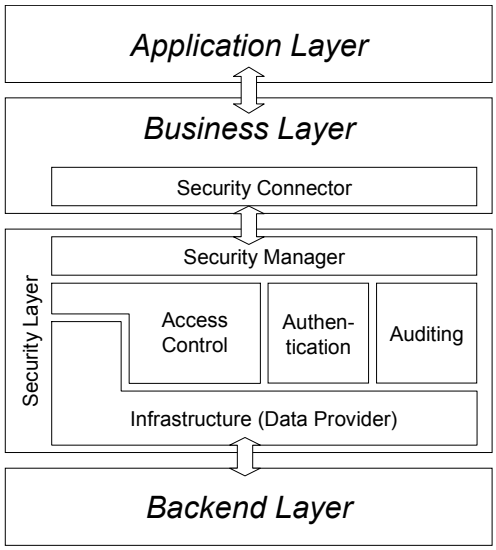


Figure 20: Layered architecture of GAMMA

3.3 Summary

This chapter presented the design of GAMMA, a framework that enables declarative security mechanisms in applications. The main objective of this framework is to provide reusable but flexible security components out of the box. In order to make the framework usable for various application domains, declarative security requirements must be offered that do not need a special underlying architecture or platform.

The framework consists of several reusable components that handle infrastructure and security enforcement. The overall design but also the functionality of the various framework components was discussed in this chapter. Especially the enforcement of the security mechanisms by integrating a new security layer that contains the objects to protect was presented and its realization aspects explained. Decoupling the security layer from the application allows a maximum of flexibility on the one hand but allows on the other hand to address complex security requirements of different application domains.

The next chapter discusses the realization aspects by presenting the JGAMMA reference implementation. The implementation is based on the Java platform, However, in order to prove the platform- and architecture neutrality, a second reference implementation that bases upon the Microsoft .NET framework is presented and discussed.

4 Reference Implementation

This chapter describes the realization of the GAMMA concept using the Java platform. First, realization aspects are described, illustrating how the concept was realized in Java. Second, the use of the JGAMMA reference implementation is shown by means of a demonstrator application. Finally, the architecture and platform neutrality of the GAMMA framework is proved by presenting a second reference implementation realized in the Microsoft .NET platform.

4.1 The JGAMMA Reference Implementation

The framework is separated into eight major packages and several sub packages, as shown in Figure 21. Each of these packages contain a set of classes that realize a specified task within the framework.

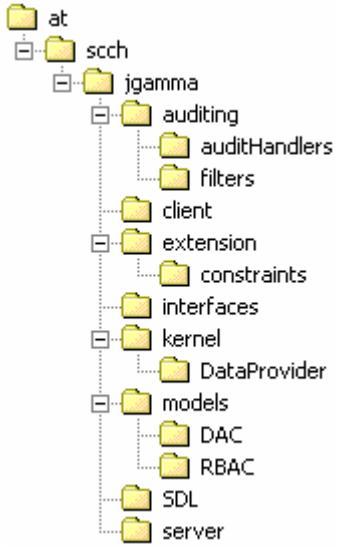


Figure 21: Package structure of the JGAMMA framework

- `auditing`: This package contains classes that are responsible for generating audit trails. Messages are sent to the auditing component which sends the message to all registered audit handlers. Before the message is stored or printed, each audit handler filters the message according to its category or priority.
  - `auditHandler`: This package contains a set of ready-to-use audit handlers.

  - `filters`: This package contains a set of ready-to-use filters for any audit handler.
- `client`: This package contains all client-side classes, thus it has to be distributed to the client that is using the framework.
- `extension`: This package and its sub packages contain extensions to the framework. Framework developers and end users can install plug-ins or extension packages here. The content of this package depends on the installed framework extensions.
- `interfaces`: This package contains all interfaces of the framework components.
- `kernel`: This package contains the core classes of the framework. These classes should not be modified in order to gain forward compatibility with future releases. However, user-defined classes can be derived from these classes and be stored in the extension or models package.
  - `DataProvider`: This package contains ready-to-use data providers for obtaining security information (SDL and object information) out of storages.
- `models`: This package contains the various security models. New models can be placed in sub packages.
  - `DAC`: All classes necessary for using the Discretionary Access Control model.
  - `RBAC`: All classes necessary for using the Role-Based Access Control model.
- `SDL`: the classes in this package build up a runtime meta model of the application (from the SDL-file) and are necessary for configuration and control of the framework.
- `server`: This package realizes the server-sided communication between client and server.

The framework itself base upon various third party libraries that provide low-level security. These libraries are located in the "ext" subdirectory (Bouncy-Castle Cryptography Library) or are part of Sun's Java 2 SDK Version 1.4.x.

4.1.1 Infrastructure Components

As mentioned earlier, the framework consists of security and infrastructure components. The infrastructure components are responsible to set up the framework and to allow communication between components within the framework and other systems. This allows the integration into existing environments by exchanging information from and to back-end systems. Furthermore, the infrastructure

components enable the use of enterprise-wide used security systems like LDAP directories or the user database in operating or database management systems.

4.1.1.1 Security Definition Language

The security definition language (SDL) forms the main part of the infrastructure components and is based on the XML standard. In fact, it is used to configure the framework, the models in use, and the security policy for a specific application. Furthermore, the SDL states which data providers are used for gathering the information for working with the framework.

The main advantage of GAMMA is the capability to use more than one security model simultaneously. The SDL defines which models are used and their order. The order is important to enable appropriate conflict resolution mechanisms if two or more models return conflicting results. Moreover, a world-assumption is assigned to each model, which can either be a closed or an open world assumption. Within the closed world assumption everything is prohibited unless explicit permissions exists, an open world assumption allows all operations unless they are explicitly prohibited.

The SDL is defined in an XML document. The structure of the document is shown in Listing 1. The italic written text is a place holder for concrete identifiers.

```
<?xml version="1.0" encoding="UTF-8"?>
<security-policy>
  <gamma application="ApplicationName">
    <models>
      <model type="model's class name"
       assumption="worldassumption"
       modelname="model's name">
        <data-provider type="provider's class name">
          <document>filename or specification</document>
        </data-provider>
      </model>
    </models>
  </gamma>
</security-policy>
```

Listing 1: SDL Sample

As one can easily see, the SDL consists of a single general tag (<security-policy>). This tag defines the scope of the security definition. Within this tag, there can be one or more instances of the application tag (<gamma application>), each specifying the security policy for a single application. The application tag requires an application name that identifies the business application. This identifier is used as a link between the security policy file and the target

application. Within this tag, the used security models are defined. This is done by specifying a model by using the `<models>` tag. Each model must have the following properties:

- `type`, specifying the Java class name that implements the model (e.g. `at.scch.jgamma.models.DAC.DACModel`),
- `assumption`, stating the model's world assumption whereas valid values are either *open* or *closed*,
- `modelname`, identifying the model within the application. The application can then query through the model's decision base via this name.

4.1.1.2 Data Providers

Data for each model is obtained from a persistent storage. These storages can differ in their structure. Furthermore, it is possible to retrieve data from various storages using so-called data providers. These data providers must be specified for each model within the model tag. Data providers are specified using a data-provider tag. Each type of data provider has its own tag (that is presented below) but they all have the same structure. The tag takes just one property, the type property that specifies the class that has to be loaded. Within the data provider tag, the document is specified that provides the connection between the data provider and the concrete storage location. The structure of the tag's content depends on the data provider (e.g., SQL statement, filename).

In order to work, a model needs following data providers:

- `ACL Data Provider` (specified via the `<acl-data-provider>` tag): is responsible for retrieving access control lists from a persistent storage. This list contains references to subjects, objects, authorizations, and constraints and has to be read first for each model.
- `Authorization Data Provider` (specified via the `<authorization-data-provider>` tag): retrieves authorization objects from a persistent storage.
- `Object Data Provider` (specified via the `<object-data-provider>` tag): retrieves the objects that have to be secured.
- `Subject Data Provider` (specified via the `<subject-data-provider>` tag): retrieves the active entities (subjects) that want to access secure objects.

- `Constraint Data Provider` (specified via the `<constraint-data-provider>` tag): retrieves constraints that restrict authorizations.

Data providers are per default bidirectional which means that they are able to write modified data back to the storage. The implementation of such custom data providers is presented in Chapter 4.3.4.

4.1.2 Client/Server Architecture

The other infrastructure components enable the connection between the application and the GAMMA framework. At runtime, the framework introduces a new security layer that holds the objects to protect. The access to these objects is regulated by the framework. Thus, it is necessary to have a connector to this layer, the so called `GAMMAClient`. Currently the communication between the application and the security layer is done using RMI. This allows the simple distribution of the application and the security components to various machines. In fact, the GAMMA framework should be installed on a trusted machine since a local installation can be circumvented by the operating system's administrator account. It is understood that appropriate measures must be taken in order to make this communication secure (e.g. use encrypted sockets for RMI). The `GAMMAClient` is the single class that has to be integrated into the business application. In fact, this component is able to perform a login to the framework and manage the user account. The login method of the `GAMMAClient` returns an interface to the security manager which forms the central part of the framework. The security manager receives all client calls and dispatches them to the appropriate security components of the framework.

The counterpart of the `GAMMAClient` is the `GAMMAServer` which resides already in the security layer. In fact, this component starts the framework by creating a security manager and sets up the communication facility (RMI) that is used by the client. The `GAMMAServer` is the bootstrap process of the framework and is detached by the security manager after the initialization completed.

4.1.3 Authentication

In order to interact with the framework, a user must be authenticated. This is automatically done by invoking the `GAMMAClient`'s `login()` method. Since the server supports multiple authentication methods, it is hard to determine for the client which method to use. The problem is caused by the fact that the client has to send different authentication data to the server, depending on the server's method. In the

case of a simple username/password authentication, the client has to send this data to the server using a secure channel. Using the secure Kerberos method, the client has to gain tickets first and send these tickets to the server whereas the tickets are generated by a server that is not part of the GAMMA framework. Currently, the method to use has to be stated in the client application by calling the appropriate login method of the `GAMMAClient` component.

However, in future releases the server will send authentication code to the client that is executed at the client side. This code then gathers the required authentication tokens and forwards them to the server. This will enable additional flexibility and protection since the authentication method can be changed anytime and the client only sends the information which is required for a correct authentication.

The authentication itself is done by the GAMMA framework. The framework provides an abstract authentication component (`kernel.Authentication`). Concrete implementations will cover different authentication methods. Currently, the framework is available with a simple `PasswordAuthentication` and a secure but more complex `KerberosAuthentication`. Each authentication component must implement the `IAuthentication` interface, that states the methods that are used by the framework to ensure the correct proof of identification of a user. Besides the pure authentication, the component is able to manage the user store that contains the various identities of the framework users. This is necessary since these storages will differ in their structure. On the other hand, delegating the management of the user store to the authentication component itself allows concepts like single sign on where the component interacts with the underlying operating system by consulting the operating system's user database and login state.

4.1.4 Auditing

The auditing component is responsible for gathering framework information and write them into a log. In fact, each framework component interacts with the auditing facility by sending messages to it. The auditing in GAMMA is highly extensible, supporting various output streams with specific filters. Thus, various `AuditHandlers` can be registered within the auditing component whereas concrete handlers enable the use of a specific output media (e.g., database, screen, logfile). When a new message arrives, the `Auditing` component dispatches the message to all registered `AuditHandlers`. Each `AuditHandler` can contain various `AuditFilters`, allowing a customization of the audit trail. Filters restrict

framework or security messages and / or the message type. This enables the usage of customized audit trails like printing only critical security messages on the server console, writing all security messages into a security log file and writing all framework relevant messages to another log file. Furthermore, this enables the integration of an *Intrusion Detection System* that acts as an `AuditHandler` and collects all messages which are evaluated and searched for attack patterns. The overall function of the `Auditing` component can be seen in Figure 22.

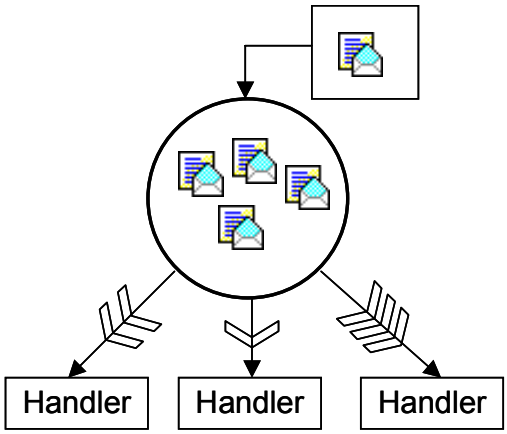


Figure 22: Auditing

In order to enable filtering mechanisms, the audit messages have a defined structure. Each message is formulated by creating an `AuditMessage` object that contains the following information:

- Message Priority: defines the type of message whereas valid options are:
  - `MSG_DEBUG`: containing a message for debugging purposes,
  - `MSG_INFORMATION`: containing a message that informs the user / administrator about something,
  - `MSG_WARNING`: containing a message that indicates that something occurred which can result in problems,
  - `MSG_ERROR`: containing a message that indicates that something went wrong, and
  - `MSG_FATAL`: containing a message that resulted from a serious misbehavior of the system.
- Message Category: defines which part of the framework produced the message whereas valid options are:
  - `MSG_FRAMEWORK`: indicating that the message affects the framework, its components, and/or the infrastructure components,

  - `MSG_SECURITY`: indicating that the message was produced by a security component (e.g. result of access checking).
- The message specified by a string.

4.1.5 Authorization and Access Control

Although authentication and auditing are required parts of a security framework, GAMMA concentrates on authorization and access control. The basic idea of how to protect business objects is to put them into a secure environment and monitor access to these objects. In fact, GAMMA intercepts each call to the protected object and forwards it to the access control engine. This interception and forward mechanism has to be done automatically, thus several components are required which enable a transparent and enforceable protection of business objects.

If a subject accesses an object in terms of calling an object's method, the invocation request is forwarded by the `GAMMAClient` to the security manager. The security manager then forwards the request to the access controller which is set up according to the current active security policy. This policy states which models are intact and their domination order. According to this domination order, the request is passed to the models by the access controller. Each model searches its decision base now for an appropriate axiom, stating whether the model grants or denies access to the object.

In the case of rule-based models (e.g., DAC, RBAC), the decision base is stated in terms of a rule base that contains access rules consisting of a subject, an object, an authorization, and optional a set of constraints. Rules can be expressed generally on whole classes, object instances, and methods or fields of an object. Classes are specified by their full qualified class name (e.g. `at.scch.object.TestClass`), object instances are identified by an object id which is assigned during the creation of the instance. Furthermore, the framework offers the possibility to assign the authorization to a specific method of an object by allowing the definition of a method signature. This enables the possibility to state access limitations at different granularity levels which raises the need for conflict resolution. A possible conflict resolution method is presented later in this chapter.

As already mentioned, the decision depends on the assigned authorization. In order to provide extensibility and flexibility, GAMMA does not provide a defined, restricted set of authorization. As a matter of fact, the authorizations know their meaning and are thus the only part within the access control that is able to state whether access should be granted or not. In order to do this, an appropriate `checkAccess()`

method must be provided that gathers information about the requested access and determines whether this access should be granted or not. This requires that the authorization component knows what it stands for and thus which information flow should be allowed or rejected. In fact, the `checkAccess()` method validates the meta information passed through the `AccessControlContext` and analyzes the request. The following contains a list of possible authorization return values. The determination of the authorization result is normally triggered by consulting all appropriate rules of a model's rule base, thus there is a tight connection between rules and authorizations.

`NO_RULE_FOUND`	This result indicates that the rule has no effect on the access decision. This case must be considered, since the rule base is not able to interpret the authorization's meaning and forwards access requests to all rules that are stated for a certain subject / object combination.
`GENERAL_RULE_AUTHORIZATION`	This result indicates that an assumption-based authorization is applicable for the access request, but the authorization is assigned on a general granularity level (class or object instance), thus it is possible that it is overruled by a more concrete rule.
`GENEARL_RULE_PERMISSION`	This result indicates that the rule fits the access request and contains a permission on a general granularity level, thus the rule grants access. However, it is possible that it is overruled by a more concrete rule.
`GENERAL_RULE_PROHIBITION`	This result indicates that the rule fits the access request and contains a prohibition, thus the rule denies access. However, it is possible that it is overruled by a more concrete rule.

EXACT_RULE_AUTHORIZATION	This result indicates that an assumption-based authorization is specified on a specific granularity level (field or method), thus overruling all general rules. Again, the result depends on the model's world assumption.
EXACT_RULE_PERMISSION	This result indicates that a permission is specified on a specific granularity level, thus it overrules all general rules.
EXACT_RULE_PROHIBITION	This result indicates that a prohibition is specified on a specific granularity level, thus it overrules all general rules.
ACCESS_DENIED	This result indicates that access is absolutely denied. This result is only returned if the access checking mechanism could not be completed due to an error or inconsistency of an authorization component.

Table 4: Possible set of authorization return values

Since various rules can return different results, a conflict resolution strategy is necessary. In fact, it is up to the model provider to define such a strategy. However, to find a trade-off between security and flexibility, we suppose the strategy of rating results. Authorizations that depend on the model's world assumption are in our opinion not as expressive as permissions and prohibitions. Therefore we give them the lowest priority. Permissions are more expressive than assumption-based authorizations but less significant than prohibitions. As a consequence, prohibitions have the highest priority. Furthermore, the result bases upon the expressiveness of the rule. As already mentioned, the framework deals with general rules and specific ones. Of course, specific rules are stronger than general rules, hence specific rules always displace general rules. This rating can also be seen from the order how the results are mentioned in Table 4.

4.1.6 Flexible Access Control

Sometimes it is not sufficient enough to simply assign authorizations to a subject / object combination. This is especially true when trying to restrict authorizations on the base of additional constraints (e.g., time, location). GAMMA enables this by allowing the optional assignment of a set of constraint components to each access rule. Again, the constraint is responsible for expressing its semantics, providing a `checkAccess()` method that returns whether the constraint grants or denies the execution of the rule. After validating the authorization positively, the model calls the `checkAccess()` method of each constraint that is defined for the rule. Other than the authorization, each constraint can only return `true` saying that the rule is valid or `false` expressing that the rule should not be considered. The rule itself is only evaluated if every specified constraint returns true.

Although the majority of access control models are rule-based, there exist other types of models that do not have a rule-base. These models must have similar mechanisms like described above. It is easily imaginable that such models consider axioms whereas these axioms are similar to rules within a rule base.

4.1.7 Security Enforcement

After describing how models find a decision whether to grant or deny a requested access operation, the question of security enforcement still remains open. In fact, the framework must provide adequate mechanisms which ensure that access requests are intercepted and forwarded to the security manager which invokes the access checking mechanism. Furthermore, it is understood that this interception must be somewhere between the client and the server.

To ensure security, the client never gets a real instance of the protected object. As mentioned above, the main idea is to keep objects in a secure place and to intercept all incoming and outgoing calls. On the other side, the client must have some objects in order to work.

A proxy controls access to an object with the help of a prefixed representative object (Gamma et al., 1995). Access to the real object is only possibly through the proxy. GAMMA protects data objects that contain sensitive information by automatically generating proxy objects and returns them instead of the real objects. Accessing these objects via the proxy directly invokes the access control mechanism because the proxy encapsulate the logic of forwarding all methods to the security manager

instead of calling the object directly. Since the real objects are kept in a separate space and can only be accessed through proxies, an application cannot circumvent the access control mechanism. However, for the application the proxy seems to be its real correspondent, meaning that an application developer does not have to make additional effort in developing his application.

```
public static Object newInstance(Object obj)
    throws InvocationTargetException {

  HashSet saveInterfaces = new HashSet();

  // add all interfaces of obj because a proxy only interacts per
  // interfaces
  addInterfaces(obj, saveInterfaces);                                    (1)

  // add the interfaces of SecureObjectProxy
  Class[] temp = new Class[saveInterfaces.size()];
  int j =0;
  for (Iterator i = saveInterfaces.iterator(); i.hasNext();) {
    temp[j] = (Class) i.next();                                          (2)
    j++;
  }

  // Create a new Proxy instance using reflection
  return java.lang.reflect.Proxy.newProxyInstance(                       (3)
    obj.getClass().getClassLoader(),
    temp,
    new SecureObjectProxy(obj));
}
```

Listing 2: Java Code that generates a proxy

Starting with Version 2, Java allows the automatic generation of proxy objects. Listing 2 shows an implementation of a generic proxy generator. The method returns a proxy to a given object instance. A proxy in Java only communicates with other classes via interfaces. Thus it is necessary that the encapsulating object does not interact with other objects by object references but by using interfaces. Furthermore, the proxy must provide the same interfaces like the encapsulating object. In a first step, the interfaces of the original object are determined using Java's reflection mechanism. This is done by calling the `getInterface()` method of the encapsulating Java class and all its super classes (see Listing 3). Since reflection only returns interface objects rather than required interface classes, these interfaces are casted to classes in a second step. The proxy is generated in a third step by calling the `newProxyInstance()` method of Java's `Proxy` class.

```
protected static void addInterfaces(Object obj, HashSet saveInterfaces)
{
  Class[] interfaces = obj.getClass().getInterfaces();
  for (int i = 0; i < interfaces.length; i++) {
    saveInterfaces.add(interfaces[i]);
  }

  Class superClass = obj.getClass().getSuperclass();
  while (superClass != null) {
    for (int i = 0; i < superClass.getInterfaces().length; i++) {
      saveInterfaces.add(superClass.getInterfaces()[i]);
    }
    superClass = superClass.getSuperclass();
  }
}
```

Listing 3: Determine an object's interfaces

```
public Object invoke(Object proxy, Method m, Object[] args)
  throws GAMMASecurityException, Throwable {

  at.scch.jgamma.interfaces.ISecurityManager securityManager =
    this.getSecurityManager();                                          (1)

  // get the subject's (caller's) identity
  IGAMMAObject identity = GAMMAClient.getIdentity(securityManager);

  String signature = null;
  try {
    // Create a signature of the method
    signature = computeSignature(m, securityManager);                   (2)
  } catch (Exception e) {
    throw new RuntimeException( ... );
  }

  // Check the access Access
  StringBuffer b = null;
  try {
    // Get the Subject represented by the account.
    String account = identity.getName();
    IGAMMAObject subject = securityManager.getSubjectByName(account);
    if (subject == null) {
      // do some audit and return...
      ...
      return null;
    }

    // Call the checkAccess remote-method of the SecurityManager
    if (securityManager.checkAccess(subject,
         (IGAMMAObject) this.secureObject, signature)) {                (3)
      return m.invoke(this.secureObject, args);                         (4)
    } else {
      // do some audit and throw security exception...
      ...
      throw new GAMMASecurityException("...");
    }
  } catch (Exception e) {
     ...
  }
}
```

Listing 4: Enforce security checks during proxy invocation

Implementing Java's `InvocationHandler` interface, the corresponding proxy object is capable to influence the method invocation of the encapsulated object. This interface requires the implementation of the `invoke()` method that is called when the proxy is accessed and directed to forward the invocation request to the real object. Thus, this is the right place to delegate the request to the security manager which in turn checks if the caller is allowed to perform the method call or not. Listing 4 shows a very simplified implementation of this mechanism.

First, the proxy gains a reference to the security manager. In a second step, the proxy generates a signature that represents the method. This is necessary since the security manager has to determine the exact method that is about to be called by the client. In a third step the access checking mechanism is invoked by calling the `checkAccess()` method of the security manager. If this method returns the value `true`, indicating that access is granted, the proxy invokes the corresponding method of the real object in a forth step.

Using Java, it is thus not necessary to provide a GAMMA *Proxy Generator*. Security is enforced by keeping objects in a secure place where the objects can only be contacted by passing the security checking mechanism.

4.1.8 Ensuring Flexibility

To provide GAMMA's flexibility, a mechanism is necessary to introspect objects at runtime. This is necessary to generate proxies as described above, and to load user-defined framework extensions (e.g., new access control models, data providers, authorizations). As already described above, the components to use are described with the SDL and can change during the lifetime of an application. In fact, the SDL states which components have to be used by specifying their class names. During the initialization of the framework, the settings are determined and the appropriate classes are loaded. The use of Java's reflection mechanism makes this easy.

Listing 5 demonstrates how a specific model is generated based on the SDL. The model's class name is passed to the method via the SDL meta data (`SDLModel`). In a first step, a reference to a Java `ClassLoader` is obtained. After loading the class (step 2), the appropriate constructor is determined (step 3), taking the arguments specified in the SDL metadata. The model is then created by calling the determined constructor (step 4). Finally, the creation process of the necessary data providers is invoked.

```
protected Model createModel(SDLModel sdlModel) {
  String modelName = sdlModel.getType();
  classLoader = this.getClass().getClassLoader();                    (1)
  try {
    Class modelClass = classLoader.loadClass(modelName);             (2)

    // get the constructor with the parameter SDLModel
    Class[] parameters = new Class[] {sdlModel.getClass()};
    Constructor constructor =
      modelClass.getDeclaredConstructor(parameters);                 (3)

    // convert the parametes of the type Class to the Type Object
    Object[] parameterlist = new Object[] {sdlModel};

    // create a new Model instance, set the DataProviders
    Model m = (Model) constructor.newInstance(parameterlist);        (4)
    // create the dataProviders
    if (m.createDataProviders(classLoader)) {                        (5)
      return m;
    } else {
      return null;
    }
  } catch (Exception e) {
    auditing.doAudit(new AuditMessage(AuditMessage.MSG_FATAL,
                     AuditMessage.MSG_FRAMEWORK,
                     "... MESSAGE ..."));
    return null;
  }
}
```

Listing 5: Dynamic creation of a model

4.1.9 Security Models

Security models form the central part of the access control engine. Since several security models exist, the framework must allow the integration of new security models. In fact, the framework provides an abstract base class `Model` that is able to perform basic operations such as managing the corresponding data providers, rule management, and the initialization process of the model-related components. Model implementations (e.g., RBAC, DAC) extend this generic base class by adding model-related semantics.

In the case of the DAC model, the semantics must be able to express the ownership paradigm. There are various ways to do this. The current implementation adds specific rules consisting of a subject, an object, and an authorization indicating that a subject owns a certain object. In fact, a new authorization, called the `OwnerPermission` is used to express this ownership paradigm. The presence of such a rule (`Subject`, `SecureObject`, `OwnerPermission`) expresses that the mentioned subject can administrate the associated object, thus the model only allows

this specific subject to add, edit, or remove rules into the rule-base for the aforementioned object.

The RBAC model requires more complex additions since a new subject-type, namely the *role*, must be added and administration aspects according to roles, user to role mapping, and authorization to role mapping must be considered. The current implementation introduces a new secure object, called `RBACAdminObject` that is able to perform administrative tasks according to role management. Allowing access to this object by assigning the `RoleManagementPermission` to a role indicates that users assigned to the related role stated in this rule are allowed to perform administrative tasks within the RBAC model. This is only one possible way, showing the various realization options for a model provider.

The implementation of new models is shown in detail in Chapter 4.3.1.

4.1.10 Separation of Duties

GAMMA provides hierarchical RBAC conforming to NIST-Standard (level 3). This includes static as well as dynamic separation of duties (SOD).

For static SOD the `RoleDataProvider` parses the RBAC security policy file, containing XML nodes that express SOD constraints. The method `isExcluded()` of the class `Role` checks the mutual exclusion of two roles. Thus, the sequence in the XML file is very important since the first role found in this file is assigned to the user and for each additional roles the SOD checks are done.

Dynamic SOD means that a user cannot activate two conflicting roles at the same time. In this case the first activated role remains active whereas the model prevents the activation of other conflicting roles.

When using the `DataProvider` one must pay attention to the order of the statements expressing these SOD constraints. The nodes `<static_SOD>` and `<dynamic_SOD>` must be stated *before* the `<role_user_assignment>` since the model must be aware of static separation of duties already before the first role is activated.

4.2 Usage of JGAMMA

This subchapter shows how GAMMA can be integrated into applications and which steps are necessary to use GAMMA. This integration will be demonstrated by an example.

4.2.1 The Vision Demonstrator

This example illustrates a simple but distributed time management system that has certain security requirements. First, the combination of different security models is shown. Second, the use of GAMMA in distributed environments using a client-server communication is presented. Furthermore, the example uses constraints rendering models inactive depending on the system time. The aim of the example is to clearly show which requirements could be addressed using GAMMA and how such a solution looks like.

During a month an employee records his activities in a company. He is allowed to grant access to other people according to his discretionary power. This requirement is best realized using a DAC (discretionary access control) model. However, there are other people in the company that are assigned to a specific task. For example, there is the secretary who needs full access to the timetable at the end of the month. Furthermore, the project manager requires read access already during the month in order to be able to track the project's progress. These requirements are best realized by using roles within an RBAC model. Thus the overall requirement is that there have to be two models active at the same time. The first model (DAC) is rendered inactive after a certain time period (at the end of the month) – at the same time all access rules must be disabled. An important requirement is that the owner should not be able to overrule the other active model in order to deny access to the secretary or to his project manager. As one can see, there are several complex requirements to the security policy.

4.2.2 Step 1: Writing the Application

To integrate GAMMA into applications, the application's design must follow some conventions. To provide protected objects, their classes must inherit from a special class named `SecureObject`. If this is not possible, these objects must at least implement the interface `ISecureObject` and provide some basic logic in their constructor (registering the resulting object into the GAMMA security layer). Since

the object is moved to the security layer and proxies are returned, an appropriate interface to the object must be provided, containing all methods that should be callable from outside. This interface must be inherited from `ISecureObject` for protection reasons, and from the `java.rmi.Remote` interface since the proxies communicate with the security layer using RMI. As a matter of fact, each method of this interface must throw a `RemoteException` (see Listing 6).

```
package at.scch.timemanagement;

import at.scch.jgamma.interfaces.ISecureObject;
import java.rmi.Remote;

public interface ICalendarPeriodObject
    extends ISecureObject, Remote {

  public void addCalendarItem(ICalendarObject item)
      throws RemoteException;

  public void removeCalendarItem(int index)
      throws RemoteException;

  public CalendarObject getCalendarItem(int index)
      throws RemoteException;

  public int getSize() throws RemoteException;

  public void setMonth(int month) throws RemoteException;

  public int getMonth() throws RemoteException;
}
```

Listing 6: Interface of CalendarPeriodObject

The current version of the framework requires `SecureObjects` and subclasses to be *remote objects*. This enables the modification of objects that are residing in the security layer because *object references* are used. Earlier versions of the framework used *object copies* rather than references, therefore the user got only copies and changes did not reflect in the framework. By defining the `SecureObjects` as remote objects, RMI stubs and skeletons have to be created for each object that is derived from the `SecureObject` base class.

Listing 7 shows an object that is used in the Vision Demonstrator and is protected by the GAMMA framework. The code additionally illustrates the specifications an object must meet. First, it is recommended that the object is derived form the `SecureObject` class. As already mentioned, the second requirement is that the object must implement an interface that contains the accessible methods. The appropriate interface is shown in Listing 6.

```
package at.scch.timemanagement;

import at.scch.jgamma.interfaces.ISecureObject;
import at.scch.jgamma.kernel.SecureObject;
import java.util.Vector;

public class CalendarPeriodObject
    extends SecureObject                                                    (1)
    implements ICalendarPeriodObject {                                      (2)

  private Vector calendarObjects;
  private int month;

  public CalendarPeriodObject(String name)
      throws RemoteException {                                              (3)
    super(name);
    calendarObjects = new Vector();
  }

  public CalendarPeriodObject(String name, String id)                       (4)
      throws RemoteException {
    super(name, id);
    calendarObjects = new Vector();
  }

  public void setMonth(int month) throws RemoteException {
    this.month = month;
  }

  public int getMonth() throws RemoteException {
    return this.month;
  }

  public void addCalendarItem(ICalendarObject item)
      throws RemoteException {
    calendarObjects.add(item);
  }

  public void removeCalendarItem(int index)
      throws RemoteException {
    calendarObjects.remove(index);
  }

  public CalendarObject getCalendarItem(int index)
      throws RemoteException {
    return (CalendarObject)calendarObjects.get(index);
  }

  public int getSize()throws RemoteException {
    return calendarObjects.size();
  }
}
```

Listing 7: CalendarPeriodObject

The last requirement is the definition and implementation of constructors that are defined for each object. Since the framework uses factory methods for creating secure objects to register them in the security layer, the framework can only create instances of *well-formed constructors* that meet the GAMMA specification. In fact, the framework specifies at least two constructors:

- The first constructor (3) takes a single name that is used for locating the object within the framework.
- The second constructor (4) additionally takes an id that is used when loading already created classes. Each object is identified via the id rather that its name, allowing to restrict access to an object instance.

If the constructor has to set properties of the class, a third constructor that takes a name, the id of the object, and arbitrary parameters has to be implemented. This constructor must then call the constructor of the base class with the parameters *name* and *id*.

In order to use the build-in data providers (e.g. the XML data provider), additional specification must be followed. The classes must be implemented as Java Beans, meaning that each field must be accessible via setter and getter methods. The data providers are only able to store fields that are related to Java standard types (primitive data types like `int`, `long`, and wrapper classes like `Integer`, `Long`, `String`). Other types require the extension of the data providers, introducing methods that are aware of how to store these types.

After creating the objects, an instance of the `GAMMAClient` class is required for each user that accesses the framework. This class holds the identity of a user so that the framework knows which user requests access. Furthermore, the `GAMMAClient` class is the connection between the client application and the GAMMA framework.

4.2.2.1 Scope of objects

When writing the application, the decision has to be made on which side (server or client) the component is actually running. Depending on this decision, the framework can access server-side components either directly, or by using a RMI reference to the server's security manager (client-side) which is of course slower that the direct reference. The following demonstrates the difference between these two access modes considering the dispatching of audit messages as example.

Server Side: Components that reside on the server side only need a reference to the security manager. Listing 8 illustrates this by calling the static method `getSecurityManager()` which returns the server's active security manager component.

```
at.scch.jgamma.kernel.SecurityManager securityManager = null;
try {
  securityManager =
    at.scch.jgamma.kernel.SecurityManager.getSecurityManager(
        "applicationname", "application.properties");
catch (Exception e) { … }
```

Listing 8: Obtaining a reference to the security manager on the server side

After obtaining the reference, all other framework components can be instantiated directly via the security manager.

```
Auditing audit = securityManager.getAuditing();
audit.doAudit(new AuditMessage(AuditMessage.MSG_DEBUG,
    AuditMessage.MSG_Framework,
    "Hello GAMMA-World!");
```

Listing 9: Use of auditing component on server side

Client Side: Each client needs a connection to the server's security manager. This reference can be obtained by using RMI communication. In order to hide the complexity, GAMMA provides the `GAMMAApplication` class that is part of the Vision Demonstrator example, but can be used in any kind of application. Listing 10 shows the implementation, describing the necessary steps in order to provide a connection between the application and the GAMMA framework.

When the class is created, it tries to obtain a reference to the server's security manager (1). This is done by using the RMI's lookup method. This method requires a URL of the server and the registration name. The URL is passed as a parameter to the constructor whereas the registration name is always `GAMMAFramework`.

If more than one GAMMA server are started on a single machine, these servers must use different registration names. In this case a customized version of the `GAMMAApplication` class has to be provided that uses the appropriate registration name.

The class provides a single method, named `login()` that handles a login-request of an application user. A user is identified via an account name and an identifier. The account name is used throughout the framework and identifies a user. The identifier may vary according to the used authentication method. In the default case this will be a password but since the parameter can take any Java object, the identifier can change depending on the authentication component.

```
import at.scch.jgamma.interfaces.ISecurityManager;
import at.scch.jgamma.interfaces.IGAMMAObject;
import at.scch.jgamma.client.GAMMAClient;
import java.rmi.RemoteException;
import java.rmi.Naming;

public class GAMMAApplication {
  /**
   local reference to server's security manager
  */
  protected ISecurityManager secMgr = null;

  /**
  Constructor: Creates a GAMMAApplication object that contains a reference
  to the server's security manager. All requests to the framework are posted
  to this object which delegates the request to the server.
  @param server Name (URL) of the server to which the client should connect.
  @exception java.lang.Exception: is raised if the creation or
   connection failed.
  */
  public GAMMAApplication(String server) throws Exception {
    try {
      // connect to the server using the name "GAMMAFramework".
      secMgr = (ISecurityManager)Naming.lookup(server +                    (1)
                "/GAMMAFramework");
    } catch (Exception e) {
      throw e;
    }
  }

  /**
  Handles the login into the framework. Each user is identified
  via an account and an identifier. The account is the name of the user, the
  identifier may change according to the used authentication mechanism
  (password, PKI, biometric token, etc.).
  @param account Accountname of user.
  @param identifier Authentication token (e.g. password)
  @return The method returns "true" if the login is completed, otherwise
  "false".
  @exception java.lang.Exception: is raised if the connection
   failed or an error occured on the server-side.
  */
  public boolean login(String account, Object identifier)
    throws Exception {
    try {
      return GAMMAClient.setIdentity(secMgr,
             (IGAMMAObject)secMgr.checkIdentifier(account,
             identifier));
    } catch (Exception e) {
      throw e;
    }
  }
}
```

Listing 10: Usage of auditing component on server side

For security purposes, some components are passed as copies instead of references. This can result in the problem that two or more instances of a component exist at the same time and that requests are not handled at the server side but rather on the object copy. Thus, the security manager provides a set of methods that forwards request to the appropriate server component. Listing 11 shows how to send an audit message to the server. Instead of obtaining a direct reference to the component, the request is

posted to a method of the security manager which in turn provides the connection to the server.

```
application.secMgr.doAudit(new AuditMessage(
    AuditMessage.MSG_DEBUG,
    AuditMessage.MSG_FRAMEWORK,
    "Hello GAMMA-World!"));
```

Listing 11: Usage of auditing component on client side

4.2.3 Step 2: Configure the Security Policy

After writing the application's classes, the security policy has to be configured according to the application's needs. This is done by modifying the SDL file. The SDL was already explained in Chapter 3.2.3. The following concentrates on the configuration of the SDL according to the demonstration application. The appropriate SDL is shown in Listing 12. The SDL file starts with the `<security-policy>` tag (1). Each entry in the file must be within the security-policy section. The tag itself contains security policies for various GAMMA applications. Each security policy is defined within the `<gamma application>` tag (2). All models must be defined within the `<models>` tag (3). The type of model to use, its assumption, and its logical name are defined using the `<model>` tag (4). Within this tag, all data providers and the model's properties are listed. Each model needs an ACL data provider (5) that provides ACL entries, an authorization data provider (7) that provides authorization components and definitions, an object data provider (8) that retrieves objects out of a persistent storage, secures them and writes them back when necessary, a subject data provider (9) retrieving a list of subjects that may access protected objects and a constraint data provider (10) that retrieves constraint objects and their definition out of a storage.

Each data provider section holds a `<document>` tag (6) that contains a description of how data can be retrieved. This tag is used by the corresponding data provider and differs from the type of data provider in use. In the example, the data providers obtain information from XML files. Thus, the `<document>` tag references to the XML file that should be loaded by the provider. Other data providers, such as database data providers, use different content or tags (e.g. an SQL statement). The set of supported tags and the required content is defined by the data provider.

```
<?xml version="1.0" encoding="UTF-8"?>
<security-policy>                                                    (1)
  <gamma application="ApplicationName">                              (2)
    <models>                                                         (3)
      <model type="at.scch.jgamma.models.DAC.DACModel"               (4)
          assumption="open" modelname="DACModel0">
        <acl-data-provider type= "at.scch.jgamma.kernel.             (5)
            DataProvider.XmlACLDataProvider">
          <document>XmlAcl_dac.xml</document>                        (6)
        </acl-data-provider>
        <authorization-data-provider type="at.scch.jgamma.           (7)
            kernel.DataProvider.XmlAuthorizationDataProvider">
          <document>XmlAuthorizations_dac.xml</document>
        </authorization-data-provider>
        <object-data-provider type="at.scch.jgamma.kernel.           (8)
            DataProvider.XmlSecureObjectDataProvider">
          <document>XmlObjects_dac.xml</document>
        </object-data-provider>
        <subject-data-provider type="at.scch.jgamma.                 (9)
            kernel.DataProvider.XmlSubjectDataProvider">
          <document>XmlSubjects_dac.xml</document>
        </subject-data-provider>
        <Constraint-data-provider type="at.scch.jgamma.             (10)
            kernel.DataProvider.XmlConstraintDataProvider">
          <document>XmlConstraints_dac.xml</document>
        </Constraint-data-provider>
      </model>

      <model type="at.scch.jgamma.models.RBAC.RBACModel"             (4)
          assumption="open" modelname="RBACModel0">
        <acl-data-provider type="at.scch.jgamma.kernel.              (5)
            DataProvider.XmlACLDataProvider">
          <document>XmlAcl.xml</document>
        </acl-data-provider>
        <authorization-data-provider type="at.scch.jgamma.           (6)
            kernel.DataProvider.XmlAuthorizationDataProvider">
          <document>XmlAuthorizations.xml</document>
        </authorization-data-provider>
        <object-data-provider type="at.scch.jgamma.kernel.           (7)
            DataProvider.XmlSecureObjectDataProvider">
          <document>XmlObjects.xml</document>
        </object-data-provider>
        <subject-data-provider type="at.scch.jgamma.                 (8)
            models.RBAC.XmlRoleDataProvider">
          <document>XmlRoles.xml</document>
        </subject-data-provider>
        <Constraint-data-provider type="at.scch.jgamma.              (9)
            kernel.DataProvider.XmlConstraintDataProvider">
          <document>XmlConstraints.xml</document>
        </Constraint-data-provider>
      </model>
    </models>
  </gamma>
</security-policy>
```

Listing 12: Demonstrator's security policy

4.2.4 Step 3: Framework Configuration

After specifying the security policy, the framework must be configured in order to work correctly. This is done by modifying some Java property files that setup the framework's components. Again, the property files depend on the set of components that are in use. The options supported by the property file depend on the effectively used components, thus the following samples are related to the default components of GAMMA.

The configuration starts with a special property file that configures the generic setup of the framework. The filename is passed as an argument when creating the framework's security manager (refer to Listing 8). It contains the setup of the security manager and the three major components of the framework, namely the access controller, the auditing component, and the authentication component.

```
SecurityManager = at.scch.jgamma.kernel.SecurityManager
AccessController = at.scch.jgamma.kernel.AccessController
AccessCtrlConfig = SDLGAMMAConfig.xml
Auditing = at.scch.jgamma.auditing.Auditing
AuditingConfig = auditing.properties
Authentication = at.scch.jgamma.kernel.PasswordAuthentication
AuthenticationConfig = authentication.properties
```

Listing 13: Framework configuration property file

The framework configuration property file (Listing 13) consists of seven lines, whereas each component, except the security manager, takes two lines. The first line points to the Java class that realizes the component's task, the second one points to the configuration file of this component. In the presented example, the standard access controller is used (`at.scch.jgamma.kernel.AccessController`). The access controller's configuration is defined by the security policy that is stated within the SDL. The SDL to use is declared in the third line within the framework property file. Auditing is also done using the standard GAMMA auditing component. The configuration is defined in another property file (`auditing.properties`). Finally, the authentication method is specified by the last two lines. In the shown example, a simple password authentication is realized.

As one can easily see, the framework configuration property file points to various other property files that handle the set up of subcomponents. Since the SDL was already described earlier, the following will concentrate on the setup of the auditing and the authorization components.

4.2.4.1 Usage of the Auditing Component

In general, the framework sends messages to the auditing component which generates an audit trail according to the defined settings. The auditing component holds a list of audit handlers that are aware of treating the messages according to their output medium. These handlers are defined within the auditing component's property file (Listing 14).

```
Handler1 = at.scch.jgamma.auditing.auditHandlers.AuditHandlerStdOut
Handler1.Properties = Handler1.properties
```

Listing 14: Content of "auditing.properties"

Listing 14 shows the definition of a single audit handler. Each audit handler needs the same two lines whereas each line has the format *key = value*. The key for an audit handler can be arbitrary but it is recommended to use the naming convention "Handler" followed by an increasing number. The second line takes a reference to a property file describing the settings of this handler. The content of such a property file is shown in Listing 15.

```
Filter1 = at.scch.jgamma.auditing.filters.AuditFilterAcceptAll
```

Listing 15: Content of "Handler1.properties"

The property file of an audit handler contains a list of filters. A message is sent to each filter before it is finally dispatched by the audit handler. Filters have the ability to remove unwanted or uninteresting messages. Each line in the file contains a reference to an audit-filter class. Again, it is recommended to use the naming convention "Filter" followed by an increasing number as the key.

4.2.4.2 Usage of the Authentication Component

Extensions to the framework allow the use of various authentication methods. Currently, the framework provides password and Kerberos authentication. The authentication method to use is defined in the framework configuration file.

Password Authentication: In order to use password authentication, the framework configuration file has to look like Listing 16.

```
Authentication = at.scch.jgamma.kernel.PasswordAuthentication
AuthenticationConfig = authentication.properties
```

Listing 16: Excerpt of the framework configuration file "application.properties"

The second line points to the property file that contains the settings of the password authentication component. The content of this property file is shown in Listing 17.

```
File = authentication.dat
```

Listing 17: Content of "authentication.properties"

The file contains a single entry pointing to the file that holds a list of the framework users and a hash-code generated from their passwords.

Kerberos Authentication: To use Kerberos authentication, a special environment offering Kerberos infrastructure is required. Various modern operating systems (e.g., Windows 2000, Solaris 8, Linux) provide such environments. The environment has to be set up before Kerberos authentication can be used in GAMMA. Furthermore, the subjects used in GAMMA must be added to the Kerberos database as principals. Listing 18 shows the necessary entries in the framework configuration file that enable Kerberos authentication.

```
Authentication = at.scch.jgamma.kernel.KerberosAuthentication
AuthenticationConfig = kerberosAuthentication.properties
```

Listing 18: Excerpt of framework configuration file "application.properties"

The second line of this configuration file points to the property file that contains the settings of the Kerberos authentication component. This property file is shown in Listing 19.

```
CallbackHandler = com.sun.security.auth.callback.TextCallbackHandler
ServerName=GAMMAServer
ClientName=GAMMAClient
# character that seperates the username from the Kerberos
# domain_realm
# depends on the platform @ is used under Windows
Delimiter=@
```

Listing 19: Content of "kerberosAuthentication.properties"

This property file contains specific entries that define the behaviour of the Kerberos system. The definition of the *callback handler* defines which method should be used when the user is prompted for typing username and password.

Kerberos is integrated using JAAS (Java Authentication and Authorization Services). Of course, JAAS must be configured in order to initiate the correct authentication method. Listing 20 shows the default configuration file for Kerberos authentication within the GAMMA framework.

```
GAMMAClient {
  com.sun.security.auth.module.Krb5LoginModule required
  storeKey=true;
  };
GAMMAServer {
  com.sun.security.auth.module.Krb5LoginModule required
  storeKey=true;
  };

other {
  com.sun.security.auth.module.Krb5LoginModule required
  storeKey=true;
  };
```

Listing 20: JAAS configuration file

Vendor specific settings for the Kerberos system must be placed in an extra file which is then passed as a VM argument when starting the GAMMA server. Listing 21 illustrates a sample Kerberos configuration file for Sun's SEAM Kerberos system.

```
 [libdefaults]
  default_realm = GAMMA.SCCH.AT

[realms]
  GAMMA.SCCH.AT = {
    kdc = authserver
    kdc = authserver
    admin_server = authserver
  }

[domain_realm]
  .gamma.scch.at = GAMMA.SCCH.AT

[logging]
  default = FILE:/var/krb5/kdc.log
  kdc = FILE:/var/krb5/kdc.log
  kdc_rotate = {
    period = 1d
    versions = 10
  }

[appdefaults]
  gkadmin = {
    help_url = http://localhost:8888/ab2/coll.384.2/SEAM
  }
  kinit = {
    renewable = true
    forwardable= true
  }
```

Listing 21: Sample Kerberos configuration file (for Sun's SEAM on Solaris 8)

4.2.5 Step 4: Staring the Application

Starting the server requires a running RMI registry. This has to be done first. Thus we propose to create a batch-file that first starts the RMI registry and afterwards the

server. It is necessary that the Java `classpath` points to all framework components that are used and to the user-defined GAMMA objects. After starting the server, various clients can be started. Both, the server and the client require some arguments that have to be passed when starting them.

4.2.5.1 Server Startup

When starting the server, a Java security policy has to be provided that allows the use of RMI. The file containing this policy is passed as a VM argument when starting Java (`-Dsecurity.policy=`*`filename`*). The server itself is started by executing the `at.scch.jgamma.server.GAMMAServer` class. This class takes two additional arguments. The first argument is the application name which is used to find the appropriate section and thus the correct setup within the GAMMA security description language. The second argument is a reference to the application's configuration file.

If Kerberos is used, additional VM arguments have to be provided. These are mainly references to the additionally needed configuration files.

4.2.5.2 Client Startup

Since the client interacts with the server via RMI, a security policy must be provided that allows RMI. Like on the server side, this policy file is passed as a VM argument when starting Java.

In order to enable the client to locate the server, the server's URL must be provided. There are two possibilities:

- Early versions of GAMMA provided this information in the file `Client.properties`. This file contains an entry `GAMMAServer=`*`url`*. When the client is started, it consults this file to obtain the server's URL.
- The second possibility is passing the URL directly as a VM argument. This is done by adding the following VM parameter `-DGAMMAServer=//`*`URL`*. The `Client.properties` file is not needed anymore.

The client is started by executing the client's Java class. Since this class is written by the application developer, required parameters depend on this class.

4.3 Extending the JGAMMA Framework

One of the most important tasks when using the GAMMA framework is the possibility to extend the framework's components. It is maybe the case that a complex application needs its own security protection mechanisms that are not already implemented in GAMMA. The framework is designed to be highly flexible and extensible. The Vision Demonstrator example presented in Chapter 4.2.1 stresses this need by having the problem of rendering the DAC model invalid after a certain time period. This problem can be solved using two different approaches. First, a time constraint can be inserted in each rule when the user creates a new rule (e.g. when an object is created or when access is granted to a third person). Since this requirement is specific to the Vision Demonstrator, this is a valid way of solving the problem. However, a second approach is to extend the framework by introducing a new model, namely the `ConstraintDACModel` class. This model automatically adds a time constraint to each rule that is created within the model. On one hand, the advantage is that the application does not have to deal with this special issue since the model fulfills the requirement. On the other hand, which is by the way the main reason why this method was chosen, this method clearly shows how the framework can be extended by a new model.

4.3.1 Implementing a New Model

The implemented model is mainly a DAC model and deals with the same issues but has a single additional requirement, namely to render rules inactive after some time period. Thus the model is derived from the existing `DACModel` class which provides a full working DAC model. Thus, extending the model means to overwrite or add new methods that cover the additional requirement.

Listing 22 shows code extracted from the `ConstrainedDACModel` class. A new model must inherit from the `Model` class (residing in the package `at.scch.jgamma.kernel`) or one of its subclasses. Since we extend the DAC model, we decided to inherit our new model from the existing `DACModel` class (1). By convention, the constructor for each model has to take an instance of the `SDLModel` class as parameter (2). This parameter represents the connection to the SDL and its meta model. In fact, the meta model influences the creation and settings of the model component.

Inheriting from an existing class or the abstract base class defines which methods have to be overwritten and which methods should be replaced by the new model. In

the case of the presented example, only the `addRule()` method has to be overwritten (3). This method is originally defined in the `Model` class and already overwritten in the `DACModel` component. Within this method, the logic of the new mechanism is implemented by adding a time constraint to each rule that is added to the model. Some code snippets are shown in Listing 22 that are often used. First, it is shown how the identity of the *internal* user (the user that starts the server process) is determined. This identity is obtained by the security manager requesting the user identity from the authentication component (5). The code shown here does not work in distributed environments where multiple users access the framework. Determine the user's identity in a distributed environment is shown in Listing 23.

Another important fact is that models or other framework components always receive proxies to protected objects. In order to evaluate or work with objects, these have to be resolved first within the server. The resolving mechanism uses the corresponding data provider to obtain a real object reference (6). This mechanism only works on the server side since clients do not have a reference to the various data providers for security purposes.

Below, the logic of the method is implemented. Within the aimed DAC-like model, only owners of objects are allowed to modify the access privileges. Thus, it must be checked if the current identity is really the owner of the object (7). If so, the subject is allowed to perform several things, in the presented case a date and time-based constraint (8) is automatically generated and added as a constraint to the created access rule (9).

The mechanism of auditing is shown in (10). A reference to the audit component is obtained by requesting it from the security manager (4).

```
public class ConstrainedDACModel
    extends DACModel {                                                    (1)

  public ConstrainedDACModel(SDLModel sdlModel) {                         (2)
    super(sdlModel);
  }

  public boolean addRule(IGAMMAObject subject,                            (3)
      IGAMMAObject secureObject,
      Object methodField,
      IAuthorization authorization,
      IConstraint constraint) {

    audit =                                                               (4)
        at.scch.jgamma.kernel.SecurityManager.
        getSecurityManager().getAuditing();
...
    IGAMMAObject identity = null;
```

```
// If this method is invoked by the server get the account using
// the SecurityManger (Remote interface).

identity = (IGAMMAObject)                                          (5)
    at.scch.jgamma.kernel.SecurityManager.
    getSecurityManager().getIdentity();

// get the SecureObject out of the SecureObjectProxy
String id = secureObject.getID();                                  (6)
IGAMMAObject obj = this.getObjectDataProvider(
).findObject(id);

// Find a rule specifying the ownership of the subject to the
// SecureObject obj.
if (isOwner(identity, obj)) {                                      (7)
  String monthName = "";
  int month = -1;

  if (obj instanceof CalendarPeriodObject) {
    if (obj.getName().indexOf("January")==0) {
      monthName = "January";
      month = Calendar.JANUARY;
    } else if (obj.getName().indexOf("February")==0) {
      monthName = "February";
      month = Calendar.FEBRUARY;
    }
    ...
    else if (obj.getName().indexOf("December")==0) {
      monthName = "December";
      month = Calendar.DECEMBER;
    } else {
      return false;
    }

    // Create time-based constraint (for current month only!)
    DateTimeConstraint c = (DateTimeConstraint)                    (8)
      this.insertNewConstraint(
        "TimeConstraint-"+monthName,
        "at.scch.jgamma.extension.constraints. "+
        "DateTimeConstraint");

      if (c == null) {
        c = (DateTimeConstraint)
          this.getConstraintDataProvider(
          ).getConstraintByName("TimeConstraint-"+
          monthName);
      }

      Calendar cal = new GregorianCalendar();
      cal.set(Calendar.MONTH, month);
      c.setStartDate(cal.get(Calendar.YEAR),
          month,
          cal.getActualMinimum(Calendar.DAY_OF_MONTH),
          cal.getActualMinimum(Calendar.HOUR),
          cal.getActualMinimum(Calendar.MINUTE),
          cal.getActualMinimum(Calendar.SECOND));

      c.setEndDate(cal.get(Calendar.YEAR),
          month,
          cal.getActualMaximum(Calendar.DAY_OF_MONTH),
          cal.getActualMaximum(Calendar.HOUR),
          cal.getActualMaximum(Calendar.MINUTE),
          cal.getActualMaximum(Calendar.SECOND));
```

```
          if (super.addRule(subject,
              obj,
              methodField,
              authorization,
              constraint)) {
            // add automatically time constraint
            return super.addRule(subject, obj,
                methodField,
                authorization, c);                                    (9)
          } else {
            return false;
          }
        } else {
          // in our case return false because we want only
          // CalendarPeriodObjects in the rule base. If other
          // objects are supported, call here addRule!
          return false;
        }
      } else {
        audit.doAudit(                                                (10)
          new AuditMessage(AuditMessage.MSG_ERROR,
              AuditMessage.MSG_SECURITY,
              this.getClass() + ".addRule => " +
              identity.getID() + " does not have " +
              "OwnerPermission for " +
              obj.getID()));
        return false;
      }
    } catch (Exception e) {
      audit.doAudit(
        new AuditMessage(AuditMessage.MSG_ERROR,
            AuditMessage.MSG_FRAMEWORK,
            this.getClass() + ".addRule" +
            e.fillInStackTrace().toString()));
      return false;
    }
  }
}
```

Listing 22: Code extract from ConstrainedDACModel

As mentioned above, the presented mechanism to obtain the user's identity only works on the server side. Each method that needs the user's identity is offered twice within the framework, with an additional set of parameters. Listing 23 shows the part of obtaining the client's user identity within the `addRule()` method.

```
public boolean addRule(IGAMMAObject caller,                           (1)
    IGAMMAObject subject,
    IGAMMAObject secureObject,
    Object methodField,
    IAuthorization authorization,
    IConstraint constraint) {

  if (isOwner(caller, obj)) {                                         (2)
    ...
  }
}
```

Listing 23: Determining a user in a distributed environment

As one can see, the method has an additional parameter indicating the *caller* of the routine (1) that represents the client user's identity. Since the identity is already resolved it can be directly integrated in security checks as this is shown in (2). Offering two methods is necessary since the latter method enables the framework to hide the complexity of RMI and network communication from the end user.

4.3.2 Implement a New Constraint

Often application developers have to deal with the challenge that access should generally be granted but additionally restricted – depending on the current time, the location of the user, or other properties. In GAMMA, access privileges are generally defined using access rules in the form of a subject, object (optional, an object's method can be stated), and authorization tuple. Each tuple defines that a subject has a certain authorization on a specified object (or a method of an object). Additionally, GAMMA offers the possibility to restrict each rule by a set of constraints. Constraints decide whether an access rule is currently valid and thus should be considered or not.

Users can implement constraints according to their needs. In the following, the realization of the date- and time-based constraint is shown that is used in the Vision Demonstrator. A code extract of this constraint class is illustrated in Listing 24.

Each constraint must inherit from the `Constraint` base class that is located in the package `at.scch.jgamma.kernel` (1). In general, a constraint must implement at least two constructors and the `checkAccess()` method according to the framework's conventions – but additional constructors with user-defined parameters are allowed (see Chapter 4.2.2). The first constructor takes a single parameter that is a string indicating the constraint's name (2). Each constraint is identified within the framework via a unique id. When calling the constructor (2) a new id is computed and assigned. However, when loading the constraint from a persistent storage, the existing id should be assigned, thus there exists a second constructor that takes the name and the id of the constraint (3). These two constructors must be specified. However, additional constructors that contain user-defined parameters can be implemented, yet some GAMMA design guidelines are followed. It is required that the first parameter must be the name of the constraint. The second parameter can be either the first user-defined parameter (4) or the constraint's id (5). Since the constraint's id is of the type string, the restriction applies that there cannot be a user-defined constructor that takes a string as its single user-defined parameter.

```
public class DateTimeConstraint extends Constraint {                (1)

  protected Date startDate;
  protected Date endDate;

  public DateTimeConstraint(String name) {                           (2)
    super(name);
  }

  public DateTimeConstraint(String name, String id) {                (3)
    super(name, id);
  }

  public DateTimeConstraint(String name,                             (4)
      Date startDate,
      Date endDate) {
    super(name);
    this.startDate = startDate;
    this.endDate   = endDate;
  }

  public DateTimeConstraint(String name,                             (5)
      String id,
      Date startDate,
      Date endDate) {
    super(name, id);
    this.startDate = startDate;
    this.endDate   = endDate;
  }

  /**
  * Method used by the Rulebase to determine whether access should be
  * granted or not.
  * @param obj Object to be accessed.
  * @return true if access can be granted, false if not.
  */
  public boolean checkAccess(Object obj) {                           (6)
    Date currentDate = new Date(); // get current DateTime

    if ((startDate == null) || (endDate == null))
      return false;

    if ((currentDate.after(startDate)) &&
        (currentDate.before(endDate))) {
      // between start- and endDate, thus allow access.
      return true;
    } else {
      return false;
    }
  }
...                                                                  (7)
}
```

Listing 24: DateTime Constraint

When an access is checked, each constraint that is assigned to a valid rule is contacted. In fact, the constraint's `checkAccess()` method is called (6). This method has to determine, whether access to an object is granted or not. The `checkAcces()` method realizes the logic of a constraint and is the only part in the constraint that knows what the constraint means to the framework. The method

receives the object that is to be accessed as a parameter and can make object-dependent decisions.

In the presented example, the constraint defines a validity period. This period has a start and an end date. Within the `checkAccess()` method, these dates are compared to the current server date and access is granted if the current date is between the validity period. The `checkAccess()` method returns the Boolean value `true` if the access rule applies. If the constraint decides that the access rule does not apply (e.g. because the time has expired), it returns `false`.

As mentioned above, each constraint of a rule is contacted. Only if *all* constraints return the value `true`, the rule applies. Furthermore, constraints can also restrict other kinds of authorizations like prohibitions or access denials (e.g. access to an object should be denied from 8:00pm to 10:00pm).

4.3.3 Implementing a New Authorization

Similar to constraints, the authorization set can also be extended with user-defined authorizations. An authorization defines a privilege that a subject has on an object. The type of privilege is specified in the concrete authorization component. Furthermore, one has to decide if an authorization should permit something (permission) or should prohibit something (prohibition). Sometimes, the decision depends on the model's world assumption saying that a privilege in an open world assumption should permit something and the very same privilege should allow something in a closed world assumption. These authorizations are called *assumption-based authorizations*. The realization aspects are explained later.

First, the implementation of an authorization that specifies the privilege to grant access to an object to other users within a DAC model is presented. Since this task can be only performed by an *owner* of an object, this authorization is called owner permission. Listing 25 shows how this permission can be implemented.

Although it seems that the owner permission grants full access to the object, this permission *only* states that a subject is allowed to manage the authorization assignment to other subjects on the specified object.

Each authorization component must inherit from the abstract base class `Authorization` (1) that is located in package `at.scch.jgamma.kernel`. By convention, authorizations must have at least two constructors. The first constructor

takes a single parameter containing the name of the authorization component (3). When a specific id has to be assigned to the component, the second constructor is issued which takes the name and the id as parameters (4). In the presented example a customized constructor is provided (2) that calls the parent's constructor with the authorization's specific name.

```
public class OwnerPermission extends Authorization {                    (1)

  public OwnerPermission() {                                             (2)
    super("OwnerPermission");
  }

  public OwnerPermission(String name) {                                  (3)
    super(name);
  }

  public OwnerPermission(String name, String id) {                       (4)
    super(name, id);
  }

  /**
  * Checks if the access for a Subject, a SecureObject a Method
  * or Field (stored in the Object context) is allowed for an
  * OwnerPermission object.
  * @param soa A SOATriple containing subject, object, authorization
  * @param context The data necessary for the access check.
  * @return GAMMADefinition.ACCESS_GRANTED because if a Rule is found
  * for the Subject to the SecureObject with an OwnerPermission the
  * access is granted independent on the world assumption of the
  * model.
  */
  public int checkAccess(Object soa, Object context) {                   (5)
    return GAMMADefinition.ACCESS_GRANTED;                               (6)
  }
}
```

Listing 25: Permission indicating ownership privilege

When access checking is done, a valid subject-object-authorization triple is located and evaluated. Since the access checking mechanism does not know what the user-defined authorizations allow, it contacts the corresponding authorization component to ask, whether access should be granted or not. This is done by invoking the component's `checkAccess()` method (5), passing the subject-object-authorization triple and the access control context that contains additional information. These parameters can be taken when more complicated access control decisions have to be made. The `OwnerPermission` simply states that a certain subject has ownership privilege over a specified object. Thus, if there exists an entry indicating a subject-object-*OwnerPermission*, the subject is *owner* of the object.

Since the owner privilege depends on the presence of the `OwnerPermission`, the implementation is rather easy and only grants access (6). More complicated

authorizations will have a more complex algorithm to determine whether access should be granted or not. However, the authorization must always return a value that indicates whether access can be granted or not. Authorizations can return various values, depending on how exact the rule is that was found (see Table 4). It is the task of the authorization programmer to determine which result should be returned.

The result of the authorization is determined by the access control model. Thus, the result is not mandatory by default. Using GAMMA's standard conflict resolution, other models are contacted if a weak result (no rule found) is returned. Strong results do not consider the opinion of other models but directly return access decisions. Furthermore, rules can be stated on exact or general level whereas general rules are overridden by exact authorizations.

4.3.4 Implementing a New Data Provider

On the one hand, new components (such as Constraints, Authorizations) need appropriate data providers since their data must be storable within the SDL. On the other hand, one might use other storage techniques (such as databases or LDAP directory). This can be realized by extending the framework with new data providers. Furthermore, each category of framework components that can be retrieved from a storage has its own data provider base class. This section shows the implementation of an XML-based authorization data provider (Listing 26).

Each data provider must inherit from the corresponding base class (1). The base class is located in the `at.scch.jgamma.kernel.DataProvider` package and is capable of retrieving a certain type of security information:

- `ACLDataProvider`: Retrieves Access Control Lists data (subject-object-authorization-constraint relations).
- `AuthorizationDataProvider`: Retrieves authorization objects.
- `ConstraintDataProvider`: Retrieves constraint objects.
- `ObjectDataProvider`: Retrieves objects containing data that is protected by the framework.
- `SubjectDataProvider`: Retrieves subjects that are active entities within the security system.
- `SecurityDataProvider`: This special data provider reads, writes, and manages the SDL and creates the models and their five other data providers mentioned above. The security data provider is the only data provider that should not be extended or overwritten by framework users!

Like all other components, also constructors of data provider must follow a framework convention. The constructor of a data provider takes a single parameter of the type `SDLDataProvider`, containing the SDL's meta information (e.g., document name, SQL-statement) concerning the data provider (2).

```
public class XmlAuthorizationDataProvider
    extends AuthorizationDataProvider {                                    (1)

  public XmlAuthorizationDataProvider(
      SDLDataProvider sdlDataProvider) {                                   (2)
    super(sdlDataProvider);
  }

  /**
  * Writes the xml file containing all Authorizations.
  * @return true if the writing succeeds otherwise false.
  */
  protected boolean writeAuthorizations() {                                (3)
    this.createDomTree();
    String fileName =
        super.sdlDataProvider.getProperty("document");                     (4)
    return this.writeDocument(fileName, document);                         (5)
  }

  /**
  * Parses the xml File containing data for the Authorizations
  * and creates a DOM-Tree, which is saved in the Document document.
  * This document is recursively traversed by the method
  * traverseTree, where the Authorizations are created.
  * @return true if the reading succeeds otherwise false.
  */
  protected boolean readAuthorizations() {                                 (6)
    ...
  }

  /**
  * Traverses the DOM-Tree and creates the appropriate Authorization
  * using data in the DOM-Tree and insert it in Authorization.
  * @param n The actual node of the DOMTree to traverse.
  */
  protected void traverseTree(Node n) {                                    (7)
    ...
    Authorization authorization =
        (Authorization) createAuthorization(name, id);                     (8)
    if (authorization != null) {
      authorizations.put(authorization.getID(),                            (9)
          authorization);
    }
  }
}
```

Listing 26: DateTimeConstraint's data provider

Data Providers are able to write / read objects to / from a storage. Thus some methods have to be overridden which are declared as abstract in the base class. In the example, the `writeAuthorization()` method that writes objects into the storage (3) and the `readAuthorization()` method that reads objects form the

storage (6) have to be overwritten. The content of these methods depends on the type of data provider.

However, it is often necessary to access the SDL's meta information within the methods (see (4)). Within the example, the information stored in the `<document>` tag of the data provider definition is obtained (see also contents and structure of the SDL).

When reading objects from the storage – as done in the example by traversing the XML DOM-tree (7) – these objects have to be created and registered. Each data provider holds a registry (a list) of registered objects it is responsible for. This registry is realized in the base class of the data providers. When an object is read, it has to be created using a factory method (8) that takes the name and id of the object that is created. If the creation process succeeds, the object has to be registered in the data provider's registry (9). The registry itself is a `Hashtable` using the object's id as a key and the object itself as value. This registry is also called the transient object storage.

4.3.5 Implementing New Auditing Components

Using auditing, the framework tracks activities within the framework system. The framework has a central auditing component where various audit handlers can be registered. When a new audit message arrives, the audit component sends the message to each audit handler that can decide if it processes the message or not. The decision is made by registering various filters within the audit hander. The audit component can be extended by implementing new handlers (that handle new output media) and new filters.

4.3.5.1 Adding a New Audit Handler

Listing 27 shows the implementation of a simple audit handler that prints the messages on the console. When implementing a new audit handler, only a few things have to be done. First, the handler must inherit from the `AuditHandler` base class that can be found in the package `at.scch.jgamma.auditing` (1). Second, constructors and methods have to be implemented. The constructor does not take any parameters and is created by the framework (2). The only method that has to be implemented is the `doAudit()` method (3) that takes a message as its single parameter and prints it on the output media.

```
public class AuditHandlerStdOut extends AuditHandler {            (1)

  public AuditHandlerStdOut() {                                    (2)
  }

  public void doAudit(AuditMessage msg) {                          (3)
    System.out.println(msg.toString());
  }
}
```

Listing 27: Simple Audit Handler

The writer of an audit handler does not have to cope with the problem of filtering messages since this is done in the base class (`AuditHandler`). The definition which audit handler should be used and the assignment of filters to audit handlers is done via the framework configuration (see Chapter 4.2.4).

4.3.5.2 Adding a New Audit Filter

Audit filters allow the customization of the `AuditHandler` according to the user's needs. Listing 28 shows a filter that accepts only messages that are classified as security-relevant audit trails.

Each audit filter must inherit from the `AuditFilter` base class that is located in the `at.scch.jgamma.auditing` package (1). Like the audit handler component, only the standard constructor (constructor with no parameters) is allowed.

```
public class AuditFilterAcceptSecurity
    extends AuditFilter {                                          (1)

  /**
  * Method, that decides if the message should be removed according
  * to the filter or not. If the message has to be removed, the
  * method returns true, otherwise false.
  * @param Message that can be filtered.
  * @return true if message should be removed, otherwise false.
  */
  public boolean filtered(AuditMessage msg) {                      (2)
    if (msg.getMessageCategory() ==
        AuditMessage.MSG_SECURITY) {                               (3)
      // do not filter message
      return false;                                                (4)
    } else
      // filter message (remove it!)
      return true;                                                 (5)
  }
}
```

Listing 28: Filter that accepts security messages only

When filtering is done, the handler calls the `filtered()` method (2) to determine if a message should be removed (filtered) or not. The method takes the message as a parameter and can determine the filtering decision according to the message's properties. In the presented example, only messages are accepted that are categorized as security relevant information (3). The method can return two values. If the message should be passed to the audit handler for output reasons, the method returns `false` which indicates that the filter does not remove the message (4). If the filter decides to remove the message, the method returns `true` (5).

4.3.6 Modifying Existing Components

It is often necessary to modify existing components. In the current version of the framework, the communication between clients and the server is done using RMI. Objects are never passed to the client, instead a reference to a proxy is passed that represents the protected object. Accessing the object is only possible via the proxy and accessing the proxy results in an access check.

In principle, framework components can be arbitrarily replaced as long as they implement the correct interfaces. However, framework developers must pay great attention to the distribution aspect since the clients interact with the framework via RMI. Thus, objects that should be passed per reference must be declared as remote, requiring adequate stub and skeleton classes. Without these classes, RMI and the framework cannot work with object references. Furthermore, developers must always return proxies to the protected objects rather than references.

In order to remain compatible with future releases of the framework, we recommend to extend the framework only at the defined places.

4.4 The GAMMA.net Reference Implementation

One of the most important design goals was to establish a platform- and architecture neutral framework. As mentioned in Chapter 3, this means that the GAMMA framework can be realized in various programming languages. In order to prove this statement, the core of GAMMA was ported to the Microsoft .NET platform. The framework uses existing security mechanisms, bases on them, and provides the same declarative security mechanisms and models like the Java reference implementation.

GAMMA was not completely ported but a feasibility study was started showing that the core concepts are realizable in .NET. Due to the design of .NET slight differences

exist compared to the Java version that affects the framework's core classes, and additional mechanisms are possible that extend the framework's functionality.

4.4.1 Compatibility Classes

In order to make the port easier, a new namespace was introduced that contains a set of *helper-classes* which re-implement some Java mechanisms. These helpers are now presented.

4.4.1.1 ClassLoader

Both, the Java and the .NET implementation must use reflection mechanisms in order to provide the necessary flexibility. In Java, loading a class is very easy by telling the `ClassLoader` which class to load. The single requirement in Java is that the target class must be locatable via Java's `classpath`. In .NET, the reflection mechanism requires additional information concerning the loaded assemblies. In fact, the user must look for the class to load (called *Type* in .NET) in each active assembly requiring an iteration process. Listing 29 shows the process of loading a class in .NET whereas the Java counterpart is illustrated in Listing 30.

For simplicity, the GAMMA.net implementation provides a `ClassLoader` component that behaves like the Java class loader.

```
public static Type LoadClass(string className)
{
  // get all active assemblies
  Assembly[] asms = AppDomain.CurrentDomain.GetAssemblies();
  Type type = null;

  // look for type definition in all assemblies
  foreach(Assembly asm in asms)
  {
    try
    {
      type = asm.GetType(className);
      if (type != null)
      {
        break;
      }
    }
    catch
    {
      // Not found in this assembly, try next one.
    }
  }
  return type;
}
```

Listing 29: Loading a class dynamically in .NET

```
public static Class loadClass(string className) {
  // get the class loader
  ClassLoader classLoader = this.getClass().getClassLoader();
  try {
    return classLoader.loadClass(className);
  }
  catch (Exception e) {
    ...
    return null;
  }
}
```

Listing 30: Loading a class dynamically in Java

4.4.1.2 Properties

The Java version of the framework is mainly configured using Java properties files. Such properties files do not exist in .NET out of the box. Thus a `Properties` component was implemented that allows the configuration of the .NET components using the same file format like the Java properties files. This allows the straight forward usage of the JGAMMA's configuration files for the GAMMA.net implementation.

4.4.1.3 StringTokenizer

The framework, in special the rule base, stores a lot of meta information concerning access checks in the form of strings. A good example is the definition of a rule that protects a method of an object. The method is specified by a signature, defining the method's name and its parameter. This information is necessary since various methods with the same name but different parameters can exist due to method overloading. The method information is stored as a string in the rule base. During the access checking mechanism, the rule's information must be compared to the access request. In order to do this, the method's signature must be parsed and the tokens must then be compared. Java offers an easy to use `StringTokenizer` which performs this task. Within .NET, tokenizers exists but are complex and difficult to use. Thus, the GAMMA.net framework provides an additional helper class that re-implements the Java's easy to use `StringTokenizer`.

4.4.2 Security Components

In principle, the design prescribes the realization of the components in a concrete development platform. However, slight differences between various platforms exist due to their peculiarities. The following shows the implementation aspects of GAMMA's security components using the Microsoft .NET platform.

4.4.2.1 Security Manager

Like in Java, the security manager represents the single entry point for client applications by accepting requests and forwards them to the appropriate security components. The implementation is straight forward to the Java platform.

4.4.2.2 Auditing

The implementation of the Auditing component is also straight forward. However, using .NET makes it very easy to use the Windows event logging facility as an output media for security and framework messages. However, careful considerations concerning filters must be done since GAMMA is very verbose and produces a lot of debugging messages.

4.4.2.3 Authentication

The effort of providing authentication components depends on the concrete authentication method. Providing a simple password authentication is an easy task whereas using the more complex Kerberos authentication is quite challenging. However, .NET makes it very easy to access the security manager of the Windows operating system, allowing single-sign on, the integration into the MS Active Directory, or the Kerberos authentication that is used in Windows 2000 domains.

4.4.2.4 Access Controller

The realization of the access controller is similar to the Java implementation. Also the `AccessControlContext` that contains an access request's meta information is straight forward since this component only holds a list that is filled with information. Both, .NET and Java provide adequate collection classes out of the box that can be used for realizing this component.

4.4.2.5 Model

In principle, the implementation of security models is straight forward to the Java implementation. However, differences exist in the related data providers because in Java they create objects out of the storage and store them in the rule base. In .NET, the data provider generate objects out of the storage but store the object id as a string in the rule base. Storing a string instead of an object reference keeps the rule base small and allows quicker processing of the rules. In Java, object references are compared whereas in .NET efficient string pattern matching methods are used. This

difference results from the fact that Java uses the proxy concept to ensure security whereas in .NET other mechanisms are used that renders the proxy concept unnecessary.

4.4.2.6 Rule Base

The rules stored in the rule base are tuples in the form of a subject, an object, optionally a field or method, a certain authorization and an optional list of constraints. The .NET implementation of the rule base is slightly different to the Java implementation. In Java, generally object references are stored in the rule base. However, methods cannot be stored since the `Method` object is not serializable and the rule base resides in another layer than the objects. Thus, it is necessary to store a method signature that references the aimed method. Out of the signature the real method is located using Java's reflection mechanisms. As already mentioned, in .NET only signatures are stored. However, the invocation of the method on the client side already works with the method signature and not with a method object like this is done in Java. Thus, there is no need to locate the method object since the whole system already works with signatures instead of references.

Furthermore, Java object references are stored in the rule base. Like this is done with methods, the .NET access mechanism uses strings for specifying objects, the so called URIs. A client accesses a certain object on the server by this URI. Access decisions can now be based on these URIs which require no additional converting mechanisms and thus increase performance. When access checking is done, the client passes the URI of the requested object and the appropriate rules are investigated.

The performance is noticeable increased since the overhead of locating objects and methods is not present and the checking mechanism is replaced by efficient pattern matching mechanisms.

4.4.2.7 Security Data Provider

The security data provider reads an XML file that contains the SDL and thus the settings of the framework. Both, JGAMMA and GAMMA.net can use the same XML file since the file's structure is obligatory. However, differences in the implementations of the data provider result from the different XML libraries and components provided by the platforms.

4.4.2.8 Security Objects

```
protected SecureObject(string name, string id)
{
  this.name = name;
  this.id = new UniqueName(id);

  try
  {
    RemotingConfiguration.RegisterWellKnownServiceType(
      typeof(SecureObject), this.getID(),
      WellKnownObjectMode.Singleton);                              (1)
  }
  catch (Exception e)
  {
    ...
  }
}
```

Listing 31: Registering a Secure Object in .NET

The subject, secure object, authorization, and constraint component are very similar. In fact, only the secure object differs in the implementation since distribution issues must be considered. In Java, the base class (`SecureObject`) handles the registration of the secure objects and enables RMI usage (see Listing 32/(1)). .NET also requires appropriate mechanisms in order to enable .NET remoting, the distribution mechanism used in the .NET platform. The object must be derived from the `MarshalByRefObject` in order to use object references instead of copies. Furthermore, the object must be registered as a well known service type (see Listing 31/(1)).

```
public SecureObject(String name, String id)
       throws GAMMAIllegalCreationException,
              java.rmi.RemoteException {
  super(name, id);
  java.rmi.server.UnicastRemoteObject.exportObject(this);          (1)
}
```

Listing 32: Registering a Secure Object in Java

4.4.3 Security Enforcement

GAMMA protects its objects by returning a reference (proxy) to the client instead of the object. When the client accesses this reference, the access is redirected to the access controller. Java provides mechanisms that enable the automatic generation of such proxy objects. As already described above, the forwarding of the method call is intercepted and redirected to the security engine. Only if the access controller grants access, the method call is really invoked.

.NET remoting provides the more sophisticated mechanisms of message sinks. Again, automatically created proxies reside on the client side. If a client invokes a method on proxy objects, invocation messages are created and passed over the wire. In fact, the whole communication between distributed objects is done by creating and handling messages. These messages can be intercepted, analyzed, and modified by message sinks, both on the server and on the client side. GAMMA.net provides a server sink that intercepts all method calls to objects residing on the server. The sink provider first consults the access controller in order to determine whether to pass the method call request to the object or not. If the access controller grants access, the sink provider let pass the message and the method is called on the object. If the access controller denies access, the sink provider removes the message and thus the method is not invoked on the protected object.

4.5 Summary

Within this chapter, the realization aspects were shown by providing a reference implementation using the Java language. The JGAMMA framework offers various security components that can be directly and transparently used when developing Java applications. Furthermore, the underlying security policy and thus the settings of the security components can be changed anytime without having to modify the application's code. Thus, the resulting application can be adapted to the customer's specific security requirements.

The practical aspects of the framework were presented by giving an idea how the usage and integration of the JGAMMA reference implementation looks like. Thus, first the use of the standard components was shown by the Vision Demonstrator example. The special security requirements of the example were documented with the extension of the framework. Thus this chapter gives an idea which effort is necessary to provide new framework components like security models, authorizations, constraints, or data providers.

In order to prove the architecture and platform independence, a second reference implementation was presented that was realized using the Microsoft .NET platform.

The next chapter discusses GAMMA and its realization aspects in detail by comparing the initial goals with the reached result.

5 Assessment and Comparison

This chapter discusses the GAMMA framework, shows some experiences and gives an outlook for future work. In special, GAMMA is compared to existing solutions, showing the practical relevance. Furthermore, open issues are identified and realization ideas are given. Before a concluding comparison to already existing products is done, experiences made during the work are stated.

5.1 Discussion of GAMMA

This subchapter discusses the GAMMA framework. First, the work is compared to the aimed goals. Before discussing open issues in the next subchapter, a feature list of the current reference implementation is given. Some remaining work will then lead to the open issues which are presented later in this chapter.

5.1.1 Reached Goals

GAMMA aims to provide declarative security mechanisms that can be easily integrated into business applications. Moreover, due to its architecture and platform neutral concept, GAMMA can be used for various application domains. The highly extensible and flexible architecture allows the customization of the framework to the developers needs. In the following, these objectives are compared against the GAMMA framework and its realization.

5.1.1.1 Active Support for Application Developers

The success of the framework depends on the provided support for application developers. In general, application developers do not want to spend much time in security design and considerations. Actively supporting the developer means that the integration of security components is a transparent task and possible already at the early stages of the development process. GAMMA provides a set of ready to use security components for authentication, authorization and auditing. These components can be directly integrated into applications, moreover the necessary infrastructure is already provided by the framework. For standard purposes, only

minor modifications have to be done, mostly by providing the appropriate configuration (properties files). The integration is a straight forward task. In order to use the framework components, the developer only has to follow the GAMMA design guidelines which are anyway conform to the design principles of modern developing platforms. For example, JGAMMA requires the user to write Java beans which is anyway common for Java classes. Furthermore, JGAMMA requires that developers *program to an interface and not to implementations*, which is a common practice for a good software design too. Thus, an advantage of GAMMA is that it requires a good design of the target application.

The only weakness is, that both reference implementations recommend that the business objects to protect must be derived from a framework base class. Sometimes deriving from this object is not possible since many modern programming languages do not support multiple inheritance. Thus, additionally interfaces are provided which allow the interaction with the GAMMA framework but require manual effort of the application developer.

However, deriving from this base class hides the whole complex logic of declarative security mechanisms from the application developer. The application developer does not need to address certain security requirements within the code. In fact, the actual security requirements are expressed outside the code in the security policy.

5.1.1.2 Flexible Security Mechanisms

Security mechanisms must be flexible and adaptable to the target environment and the software's needs. In fact, GAMMA provides a highly flexible security infrastructure that can be adapted to any user needs. This goal is reached by strictly decoupling the security layer from the application. Since security requirements are expressed outside the code and the code itself does not contain a single security statement, the provided solution can be adapted to the customer's needs without having to touch any application code. Furthermore, the security requirements can be expressed by the customer, since the security policy can be modified at any time during the software lifecycle. This increases the maintainability and reusability of the resulting code.

5.1.1.3 Open and Neutral Architecture

One of GAMMA's design goals is to provide an open and extendable architecture. In fact, GAMMA offers a set of components that interact via defined interfaces.

Providing adequate implementations to these interfaces allows framework developers to extend the framework by introducing new components (e.g., models, data providers, subjects, authorizations, constraints) or to replace components as a whole.

During the design of the architecture, careful considerations were made that the framework does not require special architectures or platforms. This enables the framework to be adaptable to any kind of application domain (e.g., server applications, Web Services, stand-alone).

The open and independent architecture was proven by providing two reference implementations, using Sun's Java and Microsoft's .NET platforms.

5.1.2 Features Supported in the Current Release

In general, GAMMA is just a concept that enables the usage of reusable security components. Thus, the framework consists only of a kernel that provides an adequate infrastructure. The reference implementation comes with a set of features like ready to use security models, data providers for various storage formats, and other directly usable security components. The current reference implementation (Version 1) supports the following features:

5.1.2.1 Access Control Models

- *DAC (Discretionary Access Control) model*, providing ownership and delegation of authorizations.
- *Hierarchical RBAC (Role-Based Access Control) model*, conforming to NIST-Standard (level 3), supporting static and dynamic separation of duties.
- *Arbitrary role-hierarchies* with user-defined activation behavior of subordinated roles and authorization inheritance.

5.1.2.2 Advanced Access Control Mechanisms

- *Constraints* that allow a finer regulation of access regulations.
- *Time-Constraints*, restricting access axioms and rules according to the system time.
- *Mixed Authorization*, expressing positive (permissions), negative (prohibitions) and assumption-based privileges.

5.1.2.3 Authentication Mechanisms

- *Password Authentication*.
- *Distributed Authentication* featuring Kerberos authentication systems.

5.1.2.4 Framework Infrastructure

- *Flexible auditing system*, supporting various output media and allowing user-defined message filtering.
- *XML data provider*, allowing to read / store authorization data from / into XML files.
- *Security policy* expressed in XML language (SDL).
- *Support for distributed applications* (RMI or .NET remoting).

5.1.3 Features Not Yet Supported

Following features are not yet supported but are subjects for future releases:

- *Cascading revocation* of authorizations in DAC model.
- *Cardinality / conditionality constraints*.
- *Additional data provider* (e.g., ODBC/JDBC/SQL, LDAP, NTLM).
- *Additional security models* and / or extensions of current models (e.g., MAC, RBAC, role-templates, proxy roles, virtual roles, social roles).
- *Building blocks for special application domains* (e.g., Web Services, Topic Maps, Data Warehouses).
- *Intrusion Detection*.

Summing up, the objectives of the GAMMA framework were reached. Furthermore, providing two reference implementations prove GAMMA's ideas and concepts feasible. However, there are still some issues that may be addressed in the future. Thus, the next subchapter discusses some open issues and provides an outlook to future work.

5.2 Open Issues in GAMMA

Several ideas and features remain open and are not yet implemented. This subchapter discusses them and describes ideas concerning the realization aspects.

5.2.1 Authentication

The actual authentication component in the current implementation is set up by the framework configuration file. In a web environment this might be inadequate since the client does not know which method is actually used. The problem results in the fact that each authentication component awaits a semantically correct identity and identifier. The identity is the name of a subject, thus differences between various authentication methods are improbably. Conversely, the structure and content of the identifier will vary depending on the actual component. Using a password authentication awaits a simple string containing the password, Kerberos on the other hand requires a valid ticket that was generated by a trusted ticket granting server.

The problem results in the fact that the authentication mechanism can change over the lifetime of a server application without explicitly notifying the clients. Thus, some client might provide wrong authentication data resulting in errors and logon denials.

A better solution might be that the client generates a request that asks for the authentication method that is currently used. The server then generates an answer, telling the client which data is required. The client can then send the correct identifier.

5.2.2 Customized Security Models

Practical experiences show, that security models presented in scientific literature often require extensions in order to meet today's security requirements. As mentioned in Chapter 2, many software applications require a combination of DAC and RABC mechanisms, following the idea of ownership but allowing the assignment of privileges to roles or user groups.

Currently, these requirements can be met by using GAMMA's features to combine models. However, providing a role-based DAC model that allows the assignment of privileges on users *and* roles would increase the usability significantly.

This model could be realized by integrating the RBAC concepts into the DAC model. In fact this requires that the resulting model must accept two types of subjects, namely *users* and *roles*. The establishment of such a new model is a straight forward task since the required classes and components already exists, only an extension to the evaluation logic of the rule base is required.

5.2.3 Multi-level Security Models

Currently, only rule-based security models are supported. However, sometimes access decisions are made by classifying subjects and objects like in MAC models.

MAC models use axioms that state the allowed information flow instead of rules. Such models can be realized by providing an axiom-based decision engine which is the counterpart of the rule base. However, the mechanism of evaluating access is the same, since the final decision is made in the access controller component whereas each model only proposes a decision.

5.2.4 Standardized Security Language

The reference implementations of GAMMA currently use a proprietary XML file for describing the security policy. Recently, several standards bodies, including OASIS (Organization for the Advancement of Structured Information Standards), IETF (Internet Engineering Task Force) and W3C (Word Wide Web Consortium), have proposed XML-based security standards. The most relevant with respect to authorization and access control are SAML (Security Assertion Markup Language) and XACML (eXtensible Access Control Markup Language), both driven by OASIS technical committees.

SAML (Oasis, 2004) is an XML-based framework for exchanging security information about authentication acts performed by subjects, attributes of subjects, and authorization decisions about whether subjects are allowed to access certain resources or not.

The purpose of XACML (Oasis, 2003) is the definition of a core schema and corresponding namespaces for the expression of authorization policies in XML against objects that are represented in XML.

At the time starting to develop GAMMA, these standards were not available or at least not stable enough to consider integration into GAMMA. However, due to the open architecture, appropriate data providers could be easily realized allowing the integration of SAML or XACML. Especially supporting SAML is a quite interesting task since this would enable a better integration into existing security systems.

5.2.5 Performance Aspects

Initially, the idea of the reference implementations was to prove the feasibility of the GAMMA concept. Although the current version exceed this initial goal by providing already a good base for the development of secure applications, some performance aspects remains open. The security manager as the single entry point is rather a bottleneck for large-scaled applications, requiring scalability mechanisms.

Scalability can be introduced by enabling clustering of GAMMA instances. Several instances of GAMMA servers could be used, each dealing with a subset of the authorization base. For example, one might assign users to a specific server instance. However, the idea of clustering requires synchronization and replication mechanisms between the instances.

The most time consuming part during access checking is the evaluation of the rules or axioms stated in the model's decision base. Keeping the decision base small significantly reduces processing time. Ensuring that only the required rules or axioms are present in the decision base would increase the overall performance. This can be done by loading the rules or axioms that are related to a user at the login-time of this user and remove them when the user logs off.

5.2.6 Intrusion Detection

Although not implemented, the design of GAMMA allows the easy implementation of intrusion detection facilities. In fact, the framework generates a very verbose audit trail. Intrusion Detection can be realized by providing a specialized audit handler that analyzed all messages and searches for patterns that indicate a possible attack. Furthermore, these audit handlers can communicate with the framework, enabling intelligent reactions to possible attacks.

5.2.7 Graphical User Interface

Providing an appropriate security policy and framework configuration is a quite challenging and error-prone task. Mistakes in the policy and the configuration can result in security breaches and thus in financial or data losses. Thus, a graphical user interface that supports the security administrator in establishing a security policy and the developer in configuring the framework would greatly improve the usability and the overall security of the system. Currently, a set of common dialogs for rule

management and other security actions are provided within the Vision Demonstrator example (see Figure 23).

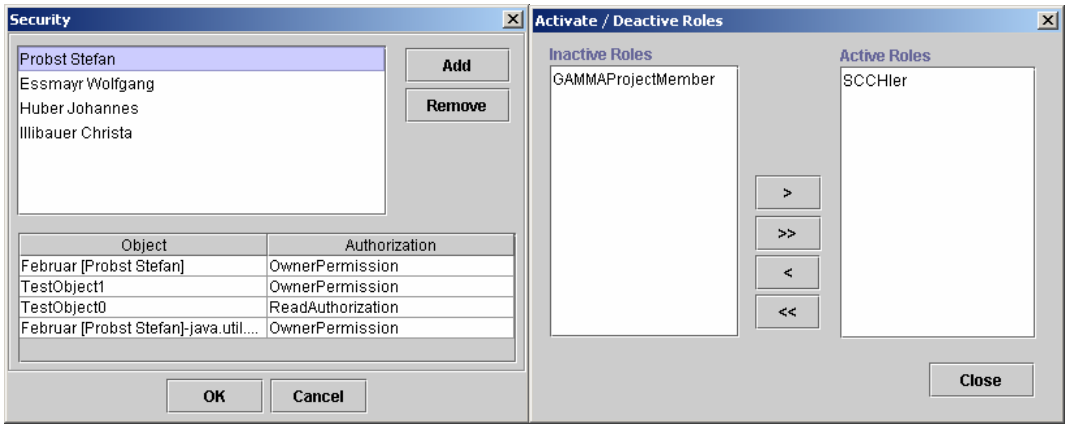


Figure 23: GUI components used in Vision Demonstrator

The provided GUI components are designed to perform modifications within the authorization base or to perform runtime actions. However, they could be easily used to build a GUI editor that allows the framework configuration and security policy management.

5.2.8 Secure Audit Trails

Currently, audit trails are stored in a file or printed on the console, depending on the audit handler in use. However, since audit trails contain security relevant information, it makes sense to protect these data. Thus, future audit handlers will be able to encrypt the generated audit trail and protect them by signatures.

5.3 Experiences

The following contains some experiences gained during the design and development of the GAMMA framework.

The first experience gained is that decoupling the security completely from the business logic is quite a challenging task. The advantages of declarative security are obvious, still many application developers strive to use programmatic security instead. The reason lies in the complexity of declarative security and the necessary effort to provide a solid base for these mechanisms. In the case of GAMMA, huge efforts were necessary to provide such declarative security mechanisms and make them as transparent as possible to application developers.

Secondly, the aim to provide a platform and architecture neutral framework requires a lot of work and a good design. Based on a good design, the implementation of JGAMMA and GAMMA.net was a straight forward process, still allowing the integration of platform specific mechanisms (e.g., JAAS, .NET remoting).

After providing the first release, efforts were made to replace RMI within the JGAMMA reference implementation. However, studies showed that the replacement is possible but increases either the overhead of communication or huge efforts with minor gains. Thus, RMI is still used in the current version of the framework.

In the case of the .NET reference implementation, some interesting experiences were made. Using the .NET remoting mechanisms open new and more efficient ways to perform security checks (e.g. by providing sink providers). Furthermore, since these messages are already in an appropriate format, more efficient methods (e.g. pattern matching) can be used to find valid rules within the rule base. Thus, we await a significant increase in performance within the .NET reference implementation. Moreover, .NET provides several security mechanisms out of the box. In special, .NET offers a ready-to-use RBAC model. First, efforts were made to integrate this model into GAMMA. However, since this native RBAC model relies and requires programmatic security concepts, the integration was not possible. Thus, the .NET reference implementation comes with its own RBAC model, which can result in misunderstandings and confusions by application developers.

5.4 Comparison

After discussion the GAMMA framework, a final comparison to similar solutions is done. The criteria catalogue for this comparison was already presented in Chapter 2. However, the following contains a short description of the criteria which are the base for the comparison.

5.4.1 Criteria Catalogue

The first criterion *Development Support* addresses the mechanisms and facilities that are offered to help developers integrating security into their application. The main focus of the presented work is to actively support application developers in writing security-aware applications. This criterion analyzes the degree of intuitive and transparent integration of security mechanisms into the application. Since this

represents the most important feature and thus is a knock-out criterion, the result can be either *yes* (developer is supported in any form) or *no*.

The criterion *Integration of Security Mechanisms into the Application* analyzes how security mechanisms must be integrated into applications. In fact, there are only two possibilities, namely *programmatic* or *declarative*.

In order to make the product usable for various application domains, the solution must be application and platform independent. Dependencies are listed within the criterion *Application / Platform Independence*.

The criterion *Expressiveness of Mechanisms* rates the offered facilities, procedures, and mechanisms that enforce the provided security. These mechanisms can help administrators in defining the security policy, provide testing facilities for the security environment, mechanisms that enforce the security policy, etc. Depending on the offered mechanisms, the products are ranked with the values *low*, *moderate*, and *high*.

Chapter 2 clearly showed the need of today's software products to support multiple security models. Thus, the criterion *Support of Multiple Models* states whether the product supports a single or multiple security models. Products can be scored with *yes* if they support multiple models, or *no* if not.

After presenting the criterion catalogue, the related work that was presented in Chapter 2 is now compared against the here presented GAMMA framework.

5.4.1.1 Development Support

The *RBAC framework for Network Enterprises* addresses only the creation of security polices but does not provide reusable components. Thus, there is no active support for developing security-aware applications. The *RBAC Implementation Project* provides a RBAC implementation for the Java platform by offering reusable components that can be used during the development of a Java application. The same can be said of the *RBAC framework using CORBA security services*. Furthermore, this solution addresses issues related to object-oriented programming and distributed software applications. *JSEF* provides a secure environment for Java code loaded from the Web. Although the architecture can also be used for other application domains, the framework does not provide mechanisms that can be integrated into applications. Using *Kava*, the developer must provide a meta-object protocol, specifying the security requirements of the application. The security is then checked

in an extra step by transforming the Java byte-code. In the case of Kava, the developer has support but the support is not really transparent and can become confusing or raise issues, especially when using COTS[2] libraries or components.

The overall design goal of GAMMA is to actively support developers in integrating security. This is achieved by providing reusable components that can be transparently integrated into applications, already at the early stages of the software development process. To provide reusability, transparent integration and high degree of flexibility, decoupling the security from the application by providing a security layer that enforces the security policy, while the code does not contain any security relevant statements.

5.4.1.2 Integration of Security Mechanisms into the Application

When looking at the integration aspects, one decides between programmatic and declarative security mechanisms. Descriptive mechanisms provide more flexibility but are harder to realize, thus often programmatic mechanisms are used.

Since the *RBAC framework for Network Enterprises* does not provide reusable components, the integration has to be done programmatically. However, it is assumable to implement declarative security components that are conform to the gained security policy, thus no clear statement concerning the integration aspect can be made. The *RBAC Implementation Project* allows the declarative usage of RBAC mechanisms within Java. Access permissions are defined using the standard Java security policy file that lies outside the application. The security mechanisms of the *RBAC framework using CORBA* are also steered by an external policy file that contains the authorization rules. *JSEF* sets up a secure environment according to an externally defined security policy. The framework itself provides only limited mechanisms for security, thus it uses declarative security. However, this security cannot be used for application development but only for setting up a secure environment. In the case of *Kava* it is not easy to clearly state whether it uses declarative or programmatic security. The security mechanisms are steered externally by the meta-object protocol which can be seen as declarative security. On the other side, the meta-object protocol must be implemented by the developer which is clearly a programmatic act.

---

[2] COTS: Commercial Of The Shelf

The advantages of declarative security mechanisms were already stressed out several times in this work. Thus, it is understood that another very important design goal of GAMMA is the use of declarative security mechanisms for providing a maximum of flexibility and reusability. In fact, the security policy of the security layer that enforces access limitations is described outside the code. The policy can be stated for each application, thus one policy can cover multiple applications and their different security requirements. Furthermore, changes in the policy influence the behavior of the security layer and thus the access checking mechanism without having to modify the application's code.

5.4.1.3 Application / Platform Independence

The *RBAC framework for Network Enterprises* aims to provide a RBAC model in a distributed environment. Thus, the model depends on the used distribution mechanism. The authors state that the most important distribution mechanisms at the time written their work were Microsoft's DCOM and CORBA. The *RBAC Implementation Project* enables the usage of RBAC in Java, thus it is restricted to the Java platform. Furthermore, the concept can only be applied to the Java platform, since Java mechanisms are taken and enhanced with RBAC mechanisms. The *RBAC framework using CORBA* requires CORBA and cannot be applied only to environments that use CORBA for distribution. *JSEF* and *Kava* are both designed for the Java platform. JSEF is further restricted only to a subset of Java since it addresses the protection and access control on mobile code. Kava requires an intermediate language that can be modified by introducing security checks directly in the already compiled code.

GAMMA is a concept for providing reusable security components. The concept does not require any special architecture or platform. In fact, during the design phase a lot of attention was paid to allow an implementation in any programming language. However, it is understood that a concrete reference implementation (e.g. JGAMMA in Java) realizes the framework components using the Java language, thus depending on the Java platform.

5.4.1.4 Expressiveness of Mechanisms

The *RBAC framework for Network Enterprises* provides expressiveness of the aimed security policy creation by allowing that the underlying security policy is formulated by the right people. Furthermore, access privileges can be assigned at various abstraction levels of the target environment. The *RBAC Implementation Project* aims

to provide a well implemented RBAC model which is reached by offering role hierarchies and separation of duties mechanisms. The *RBAC framework using CORBA* has only a moderate expressiveness in its current implementation, since only flat and hierarchical RBAC are supported. *JSEF* offers positive and negative authorizations and a hierarchical policy which allow to cover simple and complex security requirements. Since *Kava* is adding its security checks directly into the compiled code, it has a high expressiveness since mechanisms cannot be easily bypassed.

The expressiveness in GAMMA is realized by providing mature security models on the one side and protecting the objects by a secure environment on the other side. Security models provided in GAMMA are conform to existing standards and have a high degree of realization (e.g. constrained RBAC). GAMMA protects sensitive objects by moving them into a secure environment established and controlled by the GAMMA framework. Applications are getting proxies instead of the real objects. Such proxies look like the real object but contain no data. When a proxy is accessed, it forwards the access request to the corresponding, real object. GAMMA intercepts this forwarding mechanism and invokes an access checking mechanism. If access is granted by the security models, the proxy is allowed to contact the real object. This approach ensures that every access to an object must pass the access checking mechanisms of GAMMA, otherwise no data is returned.

5.4.1.5 Support of Multiple Models

The *RBAC framework for Network Enterprises*, the *RBAC Implementation Project* and the *RBAC framework using CORBA* were designed to provide RBAC mechanisms in the target environment. Thus it is understood that no other security models than RBAC are supported. *JSEF* also offers only RBAC mechanisms. *Kava* allows various security models and also a combination of them. However, they must be implemented by stating them using the meta-object protocol which can be a difficult and time-consuming task.

GAMMA offers multiple security models (e.g., DAC, RBAC) and allows any combination of them having different world assumptions. Moreover, new or customized models can be introduced by extending the GAMMA framework. Having multiple models with optionally different world assumptions raises the possibility of conflicts among these models. Thus, GAMMA also provides a conflict resolution mechanism.

5.4.2 Comparison

Criterion / Solution	RBAC for Network Ent.	RIP	RBAC using CORBA	JSEF	KAVA	GAMMA
Development Support	✗	✓	✓	✗	✓*	✓
Integration	N/A	decl.#	decl.#	N/A	decl.§	decl.
Dependencies	Distr. Mech.	Java	CORBA	Java	Java	✗
Expressiveness	high	high	moderate	high	high	high
Multiple Models	✗	✗	✗	✗	✓	✓

* not very transparent
# not very expressive
§ no clear statement possible

Table 5: Comparison of Authorization Solutions

5.5 Summary

This chapter discussed the presented framework GAMMA and compared the initial goals with the reached results. The discussion showed that the goals are reached and that it is possible to use the framework or its concepts and ideas for realizing security-aware applications.

However, still some open issues remain which are also listed in this chapter. These issues address some missing aspects of the framework. Conversely, ideas are presented that show how these issues can be realized, extending the functionality of the framework and increasing the practice relevance.

Finally, the GAMMA framework is compared against similar solutions. The comparison was made using a criteria catalogue, evaluating some functionality that is required in order to provide reusable components that can be neatly and easily integrated into software applications.

The next and last chapter contains a general conclusion, giving an overall summary of this thesis and the aspects shown, and provides and outlook that states further steps and tasks to be realized within the GAMMA framework.

6 Conclusion

This chapter contains an overall summary of the presented work here and shows the results and some experiences made during the work was done. Finally, some future work is mentioned, showing possible directions for increasing the usability and maturity level of the GAMMA framework.

6.1 Summary

Today, addressing security is an absolute need in software development. However, the most used software development platforms do not provide adequate mechanisms that allow an easy development of secure software applications. In fact, developers are forced to address security by providing special security statements that enforce a certain security policy. As showed in the introduction, this *programmatic security* comes with a lot of disadvantages. Thus, there is a need for security mechanisms that protect code and resources but which are provided and maintained outside the application.

Chapter 1 identified various goals which have to be met in order to provide a good means for developing secure software applications. According to Probst and Küng (2004), these goals can only be reached by providing *declarative security* mechanisms which allow on the one hand greater flexibility, reusability, and maintainability to the code, and on the other hand enable developers to write their code in a natural way. In fact, security is done by an *additional security layer* that holds all business objects and monitors all access requests to them. Since the business objects do not contain any security code, the reusability is significantly increased. Declarative security also stands for *flexibility*, since the security policy can be changed anytime, allowing the customer to adapt the security requirements to his site. Summing up, declarative security helps to develop secure, flexible, and maintainable software applications. The importance of declarative security mechanism as described in this work and its practical relevance can easily be seen since huge efforts are undertaken nowadays in providing such mechanisms. In fact, it is foreseeable that future releases of operating systems and programming environments will provide adequate solutions for providing declarative security.

As a matter of fact, this work presents an approach that offers declarative but expressive security mechanisms providing a range of high-level security components and models like they were described in Chapter 2, including discretionary access control (DAC), role-based access control (RBAC), and the possibility to use multiple authorization models. In order to make these components expressive enough to cover also complex requirements, the models support positive and negative authorizations and further access constraints like separation of duties or time-based access limitations.

During the work, a framework was established which aims to actively support application developers in integrating security already at the early stages of the software engineering process, whereas the usage of these mechanisms is as transparent as possible, hiding the complexity of security from the framework users. Since today's software development is faced with the problem of having different target software architectures (e.g., standalone, web-based, client/server), it was one of the framework's most important design goals to be highly flexible and adaptable. In order to fulfill these requirements, it was necessary that the framework *did not depend* on a specific architecture and platform. In fact, the framework must be realizable on various platforms. This is shown by providing two reference implementations whereas the first is implemented using the Java platform and the second is realized within the Microsoft .NET environment.

6.2 Results

As mentioned above, the overall goal of the work was to provide adequate mechanisms for realizing various security components. This work presents a conceptual framework, called GAMMA, that enables the usage of declarative security mechanisms within modern software applications. This is done by providing an infrastructure that introduce a new security layer which encapsulate the business objects to protect from the application. Transparent mechanisms ensures that the developer does not have to perform extra steps in order to rely on the offered security components.

In order to establish such a framework, declarative security mechanisms are necessary. As this thesis shows, offering such declarative mechanisms goes ahead with an enormous initial work, providing an appropriate infrastructure and security components. However, having the framework significantly relieves application developers since they do not address security requirements in special during the

coding phase. In fact, security can be done by the right people since the final security policy can be defined by the customer himself. In general, the initial costs of providing once a framework are rapidly compensated because many applications can rely on these mechanisms and the expenses for implementing these applications is reduced. Having programmatic security on the other side would mean constant expenses during application development and extra costs when maintaining or installing the application on other sites with different security requirements.

The highly flexible architecture brings another advantage since the framework can be embedded in various existing security infrastructure components (e.g., Kerberos authentication system, single-sign-on via operating system).

Having a platform and architecture neutral design enables the framework to be used in various application domains. In fact, the framework can be used in standalone applications as well as in server environments or as a base for Web Service development. As a result, the presented framework provides a solid base for today's and tomorrow's secure software development.

Finally, the feasibility of the framework was proven by providing two reference implementations. Using the Java language, the JGAMMA reference implementation enables developers to use declarative security mechanisms within the Java platform. Furthermore, the architecture and platform neutrality aspect was proven by providing a second reference implementation basing upon Microsoft's .NET technology.

6.3 Future Work

The future work separates itself into two major parts. On the one hand, currently we have to face new computer usage scenarios that come with specialized security requirements (e.g. Web Services, see Ziebermayr and Probst, 2004). On the other hand, we identified some issues on the current framework that have to be addressed in order to increase its usability.

Within the scientific area, we are currently working on testing approaches and models for security components. These approaches covers the right and thus secure usage of the framework as well as the integration and extension of new components. It is understood that introducing new components can open the risk of possible security breaches, thus we want to provide adequate testing mechanisms.

Another important issue is the integration of a standardized security language in order to make the framework compatible to other security systems. Thus, currently SAML (see Oasis, 2004) is investigated that allows exchanging security information about authentication and access control decisions.

Since currently a lot of research is done on providing new security mechanisms for specialized target environments (e.g., peer to peer networks, mobile computing, Web Services), it is desirable to extend the framework's mechanisms by providing new components or specialized security models and authorization mechanisms.

In the field of providing a mature product, there are also still some open issues which are discussed in Chapter 4.2.

In order to make the framework usable in productive server environments, some performance aspects needs to be addressed. However, Chapter 4.2 already shows solutions to the existing problems which increase the scalability and overall performance of the security engine.

Another important point is the complexity of the framework setup. In the current version, these settings must be done in various files, defining the security policy on the one hand, and the components setup on the other hand. Providing a graphical administration tool for defining the security policy and the setup of the framework would significantly increase the usability. Furthermore, most of today's security breaches result from wrong configured software programs, thus from the security perspective it is absolutely necessary to provide adequate mechanisms that support the creation and verification of the framework's configuration.

Summarizing, the work presented in this thesis allows an easy integration of reusable security components into various kinds of software applications. The framework's design and the various offered mechanisms enable the straight forward development of secure software by reducing the costs since security can be taken as granted. Although some issues remains that have to be addressed in order to use the framework in a productive environment, the mature level is very high and only minor efforts must be taken in order to provide a solid base for the development of security-aware software applications.

7 Lists

7.1 List of Figures

7.2 List of Listings

7.3 List of Tables

7.4 Literature

Ashley P., Vandequauver M. (1998): Intranet Security – The SESAME Approach. Kluwer Academic Publishing, 1998.

Beznosov K., Deng Y. (1999): A Framework for Implementing Role-Based Access Control using CORBA Security Service, *Proc. 4th ACM Workshop on Role-Based Access Control*, Fairfax, VA, USA, Oct. 28-29, 1999.

Castano S., Fugini M., Martella G., Samarati P. (1995): Database Security. Addison-Wesley, 1995. ISBN 0-201-59375-0.

Castano S., Fugini M. (1998): Rules and Patterns for Security in Workflow Systems, *Proc. 12th IFIP WG 11.3 Working Conf. on Database Security*, Chalkidiki, Greece, July 15-17, 1998.

Essmayr W., Pernul G., Tjoa A-M. (1997): Access Controls by Object-Oriented Concepts, *Proc. 11th IFIP WG 11.3 Working Conf. on Database Security*, Lake Tahoe, California, USA, Aug. 1997.

Essmayr W., Kapsammer E., Wagner R., Tjoa A. (1998) Using Role-Templates for Handling Recurring Role Structures. *Proc. 12th IFIP WG 11.3 Working Conf. on Database Security*, Chalkidiki, Greece, July 15-17, 1998.

Essmayr W., Probst S., Weippl E. (2001): A Comparison of Distributed Authorization Solutions, *Proc. 3rd Int. Conference on Information Integration and Web-based Applications & Services (IIWAS)*, Linz, Austria, Sept. 10th-12th, 2001.

Essmayr W., Probst S., Weippl E. (2004): Role-based Access Controls: Status, Dissemination, and Prospects for Generic Security Mechanisms, *Electronic Commerce Research*, Kluwer Academic Publishers 4(1), pp 127-156, Jan. 2004

Fernandez E., Nair K., Larrondo-Petrie M., Xu Y. (1996): High-Level Security Issues in Multimedia/Hypertext Systems. *Proc. IFIP TC6/TC11 Int. Conf. on Communications and Multimedia Security*, Essen, Germany, 1996.

Ferraiolo D., Kuhn R. (1992): Role-Based Access Control (RBAC), *Proc. 15th NIST-NSA National Computer Security Conf.* pp. 554-563., Baltimore, Maryland, Oct. 13-16, 1992.

Ferraiolo D., Gilbert D., Lynch N. (1993): An Examination of Federal and Commercial Access Control Policy Needs. *Proc. NIST-NCSC National Computer Security Conf.*, National Inst. Standards and Technology, Gaithersburg, Md., pp. 107-116, 1993.

Foundstone Inc., CORE Security Technologies (2001): Security in the Microsoft® .NET Framework, http://www.foundstone.com/pdf/dotnet-security-framework.pdf (last accessed on May 31., 2002).

Gamma E., Helm R., Johnson R., Vlissides J. (1996): Design Patterns. ISBN 0-201-63361-2, Addison-Wesley, 1995.

Gavrila S., Barkley J. (1998): Formal Specification for Role Based Access Control User/Role and Role/Role Relationship Management, *Proc. 3rd ACM Workshop on Role-Based Access Control*, Fairfax, VA, USA, 1998.

Giuri L., Igilo P. (1997): Role Templates for Content-Based Access Control, *Proc. 2nd ACM Workshop on Role-Based Access Control (RBAC'97)*, Fairfax, VA, USA, Nov. 6-7, 1997.

Giuri L. (1998): Role-Based Access Control in Java, *Proc. 3rd ACM Workshop on Role-Based Access Control*, Fairfax, VA, USA, Oct. 22-23, 1998.

Gollmann D. (1999): Computer Security. John Wiley & Sons, 1999, ISBN 0-471-97844-2.

Hauswith M., Kerer C., Kurmanowytsch R., (2000): A flexible and extensible security framework for Java code, *Proc. 9th International World Wide Web Conference*, Amsterdam, May 2000.

Herzberg A., Mihaeli J., Mass Y., Naor D., Ravid Y. (2000): Access Control Meets Public Key Infrastructure, Or: Assigning Roles to Strangers, *Proc. IEEE Symposium on Security and Privacy*, May 14-17, 2000, Oakland, California, USA.

Izaki K., Tanaka K., Takizawa M. (2000): Access Control Model in Object-Oriented Systems, *7th International Conference on Parallel and Distributed Systems: ICPADS'00* Workshop, IEEE 2000.

JAAS (2000): Java Authentication and Authorization Service 1.0, Developer's Guide, http://java.sun.com/security/jaas/doc/api.html (last visited Dec. 13, 2001)

Jendrock E., Bodoff S., Green D., Haase K., Pawlan M., Stearns B. (2002): The J2EE Tutorial, ISBN 0-201-79168-4, Addison Wesley, 2002.

Kabay M., Identification, Authentication and Authorization on the World Wide Web, White Paper; http://secinf.net/info/www/iaa/iaawww.shtml (last visited 30.01.2002)

Lai C., Gong L., Koved L., Nadalin A., Schemers R. (1999): User Authentication and Authorization in the Java Platform, *Proc. 15th Annual Computer Security Applications Conference*, Phoenix, AZ, USA, December 1999.

Linn J. (1997): RFC 2078 – Generic Security Service Application Program Interface, Version 2, Request for Comments 2078, Internet Engineering Task Force, January 1997.

McLean J., (1990): The Specification and Modeling of Computer Security, IEEE Computer 23(1): 9-16, 1990.

McMahon P. (1994): SESAME V2 Public Key and Authorization Extensions to Kerberos, Proc. ISOC Symposium, 1994.

Myers J. (1997): Simple Authentication and Security Layer (SASL), Request for Comments 2222, Internet Engineering Task Force, October 1997.

Nyanchama M., Osborn S. (1994): Database Security VIII: Status and Prospects, IFIP Working Conf. On Database Security, *In Proc. 15th Annual computer Security Applications Conference*, North-Holland, 1994.

Oasis (2003): eXtensible Access Control Markup Language (XACML) Version 1.1, Committee Specification, 07. August 2003, http://www.oasis-open.org/committees/xacml/repository/cs-xacml-specification-1.1.pdf.

Oasis (2004): Technical Overview of the OASIS Security Assertion Markup Language (SAML) V.1., Draft 04, 30 March 2004, http://www.oasis-open.org/committees/documents.php?wg_abbrev=security.

Oppliger R., Pernul G., Strauss C. (2000): Using Attribute Certificates to Implement Role-Based Authorization and Access Controls, *Proc. Fachtagung Sicherheit in Informationssystemen (SIS)*, Zürich, Schweiz, 5.-6. Okt. 2000.

Osborn S., Sandhu R., Munawer Q. (2000): Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies, *ACM Transaction on Information and System Security*, Vol. 3, No. 2, pp. 85-206, May 2000.

Pernul G. (1994): Database Security, Advances in Computers, Vol. 38, Academic Press, ISBN 0-12-012138-7

Probst S., Essmayr W., Weippl E. (2002): Reusable Components for Developing Security-Aware Applications, *Proc. 18th Annual Computer Security Applications Conference (ACSAC)*, Las Vegas, NV, Dec. 9-13, 2002.

Probst S., Küng J. (2004): The need for declarative security mechanisms, *Proc. 30th EUROMICRO,* Rennes, France, Sept. 2004.

Ramaswamy C., Sandhu R. (1998): Role-Based Access Control Features in Commercial Database Management Systems, *Proc. 21st NIST-NCSC National Information System Security Conference*, pp. 503-511, Arlington, VA, October 5-8, 1998.

Samar V. (1996): Unified Login with Pluggable Authentication Modules (PAM), *Proc. of the 3rd ACM conference on computer and communication security,* New Delhi, India, 1996.

Sandhu R. (1996): Role-Based Access Control Models, IEEE Computer, Vol. 29, No. 2, Feb. 1996.

Sandhu R., Coyne E. (1996): Role-Based Access Control Models. IEEE Computer, Vol. 29, No. 2. Feb. 1996.

Sandhu R., Samarati P. (1996): Authentication, Access Control, and Audit, ACM Computing Surveys, Vol. 28, No. 1, March 1996.

Sandhu R., Ahn G. (1998): Group Hierarchies With Decentralized User Assignment in Windows NT, *Proc. International Association of Science and Technology for Development (IASTED), Conference on Software Engineering*, Las Vegas, October 1998.

Sandhu R., Ahn G. (1998): Decentralized Group Hierarchies in Unix: An Experiment and Lessons Learned. *Proc. 21st NIST-NCSC National Information System Security Conference*, Arlington, VA, October 5-8, 1998.

Sandhu R., Ferraiolo D., Kuhn R. (2000): The NIST Model for Role-based Access Control: Towards a Unified Standard, *Proc. of 5th ACM Workshop on Role-Based Access Control*, July 2000.

Schier K. (1998): Multifunctional Smartcards for Electronic Commerce - Application of the Role and Task Based Security Model, *Proc. 14th ACSAC*, Scottsdale, Arizona, Dec. 7-11, 1998.

Sun Microsystems (1999): Java Security Architecture, http://java.sun.com/j2se/1.4/docs/guide/security/spec/security-specTOC.fm.html (last accessed on May 31, 2002).

Tenday J., Quisquater J., Lobelle M. (1999): Deriving a Role-Based Access Control Model from the OBBAC Model, *Proc. IEEE 8th Int. Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, Palo Alto, CA, USA, June 16-18, 1999.

Thomas R. (1997): Team-based Access Control (TMAC): A Primitive for Applying Role-based Access Controls in Collaborative Environments, *RBAC 97* Fairfax Va USA, ACM 0-89791-985-8/97/11. 1997.

Thomsen D., O'Brien D., Bogle J. (1998): Role-Based Access Control Framework for Network Enterprises, *Proc. 14th Annual Computer Security Applications Conference*, Scottsdale, Ariziona, Dec. 7-11, 1998.

Welch I., Stroud R. (1999): Supporting Real World Security Models in Java, *Proc. 7th IEEE Workshop on Future Trends in Distributed Computing Systems*, Dec. 20, 1999, Tunisia, South Africa.

Ziebermayr T, Probst S. (2004): Web Service Authorization Framework, *Proc. of ICWS*, San Diego, 2004.

Zurko M., Simon R., Sanfilippo T. (1999): A User-Centered, Modular Authorization Service Build on an RBAC Foundation, *Proc. IEEE Symposium on Security and Privacy*, Berkley, CA, USA, May 1999.

Appendix: Curriculum Vitae

Personal Data	Name Birthday Address Maritial status	Dipl.Ing.(FH) Stefan Probst 24.07.1978 in Vienna, Austria Weingarten 6 / 7 4232 Hagenberg Single
Education	07.1992 06.1996 06.2000 11.2004	Finished secondary school in Oberschützen General qualification for university entrance (Matura) at BORG Güssing with focus on informatics Diploma at the university of applied science for software engineering in Hagenberg Finished PhD studies at the Johannes Kepler University in Linz
Diploma Thesis	Topic	Study the applicability of the Java language for developing large scale software systems in the context of high-energy physics.
Dissertation	Topic	GAMMA – A platform independent framework for reusable authentication, authorization, and auditing components.
Professional Experience	08.1997 – 09.1997 08.1999 – 05.2000 09.2000 – 08.2004 starting with 10.2001 starting with 10.2003 starting with 10.2004	Internship at AUTECH GmbH, Radolfzell, Germany Pratical semester and diploma thesis at the "European Organization for Nuclear Research" (CERN) in Geneva, Switzerland Member of Scientific Staff at the Software Competence Center Hagenberg, starting with January 2001 project manager of strategic project GAMMA Lecturer at the university of applied science in Hagenberg in the fields operating systems, network technology, formal languages and compiler construction, and distributed software systems. IT-Security consulting for small and medium sized companies Microsoft High School Advisor (Western Austria)