Zás – An Aspect-Oriented Access Control Framework

Paulo Zenida

Presented to the Instituto Superior de Ciências do Trabalho e da Empresa

in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Computer Engineering and Telecommunications

July 19, 2007

Abstract

Traditionally, access control system architectures are based on the abstract reference monitor model proposed by Anderson, which tries to separate the access control logic from the logic of applications.

The implementation of this model has been difficult since access control presents itself as a crosscutting concern, i.e., it crosscuts the functionalities of applications. However, the developments of the techniques that support the separation of concerns, particularly aspect oriented programming, have been enabling the development of systems in which the access control code is not scattered through the code of the application. Nevertheless, these solutions are still too specific to a given application.

This work presents an access control framework for Java applications, named Zás, which can be reused and that applies the abstract reference monitor proposed by Anderson. This framework supports access control policies using different kinds of context information and allows them to be changed at runtime. Zás was developed in the aspect oriented programming language AspectJ and it uses Java 5 annotations.

We used Zás in several small applications since its inception, so as to gain experience and insight from its application. Nevertheless, to evaluate the performance and applicability of the final version of the Zás framework prototype, we integrated it in a very large Web application called Fénix, which brought us very interesting results, such as showing the easiness of applicating Zás to already existing applications, and also the main caveats and limitations of Zás.

Keywords: access control, authorization, aspect oriented programming

Resumo

As arquitecturas dos sistemas de controlo de acesso são tradicionalmente baseadas no modelo abstracto de monitor de referência proposto por Anderson, o qual visa separar a lógica do controlo de acesso da lógica das aplicações.

A concretização deste modelo tem sido dificultada pelo facto de o controlo de acesso se apresentar como uma faceta transversal às funcionalidades das aplicações. No entanto, os progressos verificados nas técnicas que suportam a separação de facetas, em particular a programação orientada para aspectos, têm permitido desenvolver sistemas em que o código de controlo de acesso não é disseminado pelo código das aplicações. Porém, estas soluções são ainda concretizações específicas, para uma determinada aplicação.

Este trabalho apresenta uma plataforma de controlo de acesso para aplicações Java, designada por Zás, que é reutilizável e que aplica o modelo abstracto de monitor de referência proposto por Anderson. Esta plataforma suporta políticas de controlo de acesso fazendo uso de diferentes tipos de informação de contexto e permitindo a sua alteração em tempo de execução. O Zás foi desenvolvido na linguagem de programação orientada para aspectos AspectJ e recorrendo às anotações do Java 5.

Nós aplicámos o Zás a várias aplicações pequenas desde o seu começo, de forma a ganhar experiência e conhecimento através da sua aplicação. No entanto, para avaliar o desempenho e aplicabilidade da versão final do protótipo da plataforma Zás, nós integrámo-lo numa aplicação Web muito grande chamada Fénix, a qual nos trouxe resultados muito interessantes, tais como mostrar a facilidade da aplicação do Zás a aplicações já existentes, e os principais problemas e

limitações do Zás.

Palavras-chave: controlo de acesso, autorização, programação orientada para aspectos

Contents

List of Figures

List of Tables

Acronyms

ACRSR Access Control Requirements Specification Ratio

AJDT AspectJ Development Tools

AO Aspect-Oriented

AOP Aspect-Oriented Programming

AOSD Aspect-Oriented Software Development

CORBA Common Object Request Broker Architecture

CPU Central Processing Unit

DAC Discretionary Access Control

DBMS Database Management Systems

DD Domain Driven

DML Domain Modeling Language

DOM Document Object Model

EJB Enterprise JavaBeans

FTP File Transfer Protocol

HT Hyper-Threading

HTTP HyperText Transfer Protocol

IDE Integrated Development Environment

IT Information Technologies

ITD Inter-Type Declaration

JAAS Java Authentication and Authorization Service

JAR Java ARchive

JDK J2SE (Java 2 Platform Standard Edition) Development Kit

JSF JavaServer Faces

JSP JavaServer Pages

J2SE Java 2 Platform Standard Edition

LDAP Lightweight Directory Access Protocol

LOC Lines of Code

MAC Mandatory Access Control

NOV Net Option Value

OASIS Organization for the Advancement of Structured Information Standards

OJB ObJect Relational Bridge

OO Object-Oriented

OOP Object-Oriented Programming

ORM Object-Relational Mapping

PAM Pluggable Authentication Modules

RAM Random Access Memory

RAD Resource Access Decision

RBAC Role-Based Access Control

SOA Services Oriented Architecture

SPL Security Policy Language

UML Unified Modeling Language

XACML eXtensible Access Control Markup Language

XML eXtensible Markup Language

Acknowledgements

I would like to especially thank my advisors, Manuel Menezes de Sequeira and Dulce Domingos, for their guidance, expertise, and support. They kept me on track and never let me go down. I would also like to thank José Pedro Pereira, from Linkare, who always showed interest in my research and with whom I had many interesting discussions about access control models, architectures, and techniques. To Ramnivas Laddad, though we never met personally (only through e-mail), a special thanks for his interest and motivating words when this research began and I showed him my first results. To the AspectJ community in general and to the AspectJ compiler development team, especially Andy Clement, for helping me with some hard AspectJ compiler problems, a deepest thanks.

I would also like to thank the insightful comments from the anonymous reviewers and audience from the ICSOFT 2006 conference to which I have submitted an accepted paper about Zás, since it allowed me to discuss with others the first results of my work, and especially to the anonymous reviewers from the AOSD 2007 conference to which I have submitted a paper about Zás which ended up being rejected, since those comments were really useful to improve the quality of my research. Finally, I am also very grateful to my family, especially my parents and my brother, who have always helped and supported me in the tough moments, and a very, very special thanks to my girlfriend, with whom I have learned a lot about myself and with whom I now know I can always count on.

Chapter 1

Introduction

Separation of concerns, as proposed by Dijkstra [17], is a key principle in software engineering. This principle is used for controlling the complexity of the applications and defends that, given a complex problem with different concerns, these concerns should be properly identified, each addressed separately and, finally, the respective solutions integrated to produce the final result of the solution. This principle has been pursued by access control[1] system architectures, since they are traditionally based on the abstract reference monitor proposed by Anderson [2], which in fact tries to separate the access control logic from the logic of the application.

Bearing in mind the guidelines which state that security Information Technologies (IT) systems should be developed by a combination of security engineering and software engineering [18], and that software engineers should not have to be concerned with security issues, and vice-versa, the development of an application should be separated, i.e., "the security code should not be mixed with the application code" [1].

The implementation of the abstract reference monitor has been difficult, since access control is a crosscutting[2] concern, i.e., "a security requirement that crosscuts the (business part of the)

[1] The "ability to permit or deny the use of an object (a passive entity, such as a system or file) by a subject (an active entity, such as an individual or process)." [46]

[2] "Features of a program that are orthogonal to the main decomposition of the program [13]".

application" [47].

Progresses in techniques supporting the separation of concerns, such as design patterns [49, 19, 20] and, more recently, Aspect-Oriented Programming (AOP) [26, 47, 44], enabled the development of systems in which the access control code is not scattered through the code of the application. Indeed, one of the advantages of AOP is that it enables the separation of crosscutting concerns from the main decomposition of the application. However, current access control solutions using these techniques have typically not been reusable nor generic.

This thesis presents an access control framework for Java applications, named Zás[3], mainly inspired by a proposal of Laddad to modularize Java Authentication and Authorization Service (JAAS) client code using AOP [26]. Zás controls the invocations of methods and constructors, as well as sets and gets of values of fields. Hence, Zás protects methods, and fields.

Zás supports access control policies which use different kinds of context information and enables the change of these policies at runtime. Zás' advantages stem mainly from the fact that an AOP approach is used, allowing it to address some of the problems found in the application of industry standards such as JAAS [29, 14, 34]. The framework was implemented in the AOP language AspectJ [4] and it uses Java 5 annotations to specify the application's protected objects and their access control requirements. A few important AOP notions used throughout this thesis are:

- join point – a well defined point in the execution of the program,
- pointcut – a set of join points, and
- advice – a piece of code that is weaved in all join points of the pointcut with which it is associated.

We used Zás in several small applications since its inception, so as to gain experience and insight from its application. Nevertheless, to evaluate the performance and applicability of the

[3]The Zás source code may be found in `http://sourceforge.net/projects/zas/`.

final version of the Zás framework prototype, we integrated it in a very large Web application called Fénix®[4], which was also the main source of motivation for the creation of this framework! Empirical results of the application of Zás to Fénix are presented in Chapter 5.

Throughout this thesis, we assume that the reader is familiarized with the basic concepts of AOP and AspectJ. Nevertheless, if that is not the case, we strongly recommend reading Laddad's AspectJ in Action [26].

1.1 Contributions

The main contribution of this thesis is the development of a reusable framework of aspects and classes for application-level access control enforcement that fully separates the functional code from the access control code. Zás uses Java 5 annotations, which augments the expressiveness of the source code, to specify the protected objects and their access control requirements.

Other important contributions of this thesis, though not the most relevant, were related to general contributions to the AOP and AspectJ communities, such as the detection of several bugs in the AspectJ compiler and their registration in the corresponding bug tracking system.

Similarly, the application of Zás to a project such as Fénix is also a contribution to the community, since it focuses on the application of AOP to a real, large application. Finally, after the hard technical and political task of introducing aspects to Fénix, with Andrew Clement's precious help, from the AspectJ project development team, and José Pedro Pereira from Linkare, we achieved our goal and new applications of AOP to that project arose, such as contracts checking (pre conditions, post conditions and invariant conditions).

[4] See http://fenix-ashes.ist.utl.pt/FrontPage/.

1.2 Structure

The remainder of this thesis is structured as follows. Chapter 2 briefly introduces the main access control concepts and related work. Chapter 3 presents a very simple toy application to show the access control crosscutting problem and four different approaches to solve it, ranging from Object-Oriented (OO) ad hoc solutions to Aspect-Oriented (AO) solutions based on JAAS and annotations. Chapter 4 presents the architecture and features of Zás and Chapter 5 evaluates its applicability. Finally, Chapter 6 presents the conclusions of the research and lays out possible directions for further work.

Chapter 2

Access control

Access control is a security service whose purpose it to guarantee the protection of resources[1] against unauthorized accesses. This chapter presents the main access control concepts and the main contributions and limitations related to the application of the abstract reference monitor.

2.1 Concepts

The development of an access control system implies the definition of an access control policy (rules and regulations) and its enforcement through appropriate security mechanisms. Access control models represent formally the access control policies, expressed through specific access control languages. Ponder [16], Security Policy Language (SPL) [15] and eXtensible Access Control Markup Language (XACML) [35] are representative examples of this kind of language.

Traditionally, there are two access control models:

1. Mandatory Access Control (MAC) models, in which access rules are system-wide and usually fixed. While rules can change over time, users cannot influence them. These models are commonly used in systems were rigorous access control is very important.

[1] In this work we will use the term "protected object" when referring to the resource which is subject to access control.

2. Discretionary Access Control (DAC) models give the owner of the protected object the right to determine the access control policy for that object. These systems are discretionary in the sense that a user that was given discretionary access to a protected object is capable of granting access to that protected object to other users. These models are used in operating systems like Unix, in which a user can specify the access rules to the files she owns.

Generically, a discretionary access control policy is defined by a set of authorizations, each typically defined as a tuple $(s, m, o, pred)$ stating that subject s may legitimately use the access mode m to access to the protected object o if the predicate $pred$[2] is true in the context of the access. The access mode may represent a specific operation performed with or over the object, or an abstract access mode, associated to a set of specific operations. Using abstract access modes has the advantage to decrease the number of permissions within the system, since each abstract access mode may be associated to a set of operations, instead of simply one.

The use of predicates augments the expressiveness of the authorizations, supporting a more fine-grained control of authorizations.

As an alternative to the MAC and DAC models, a model based on the notion of role, viz. Role-Based Access Control (RBAC), was proposed. This model was very well received, since the notion of roles fits well to the common notion of function in organizations. RBAC models associate authorizations to roles performed by subjects [21]. RBAC authorization $(r, m, o, pred)$ states that a subject performing role r can legitimately use mode m to access to the protected object o whenever the predicate $pred$ is true. Since subjects are not directly associated with access modes, but indirectly through the role or roles they perform, the management of individual privileges in the system is often only a matter of assigning the appropriate roles to each subject.

[2] In some models the predicate is not used and hence is assumed to be always true.

2.2 Access control system architectures

Access control system architectures are traditionally based on the abstract reference monitor proposed by Anderson [2], Figure 2.1. A reference monitor intercepts all access attempts from subjects to the protected objects. Conceptually, a reference monitor has two main functions:

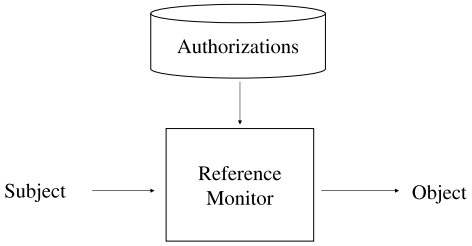

Figure 2.1: Abstract reference monitor proposed by Anderson.

1. a decider, responsible for evaluating the legitimacy of the accesses,
2. an enforcer, responsible for intercepting all access attempts and enforce the decision that was taken.

According to this model, all access attempts are intercepted by the enforcer, which asks the decider to determine the legitimacy of the access, searching the authorizations database.

Recently, the Organization for the Advancement of Structured Information Standards (OASIS) has applied this model in its proposal for the XACML access control system [35]. However, implementing this model has been difficult, since access control crosscuts applications functionalities.

Middleware systems supply a simple solution to separate access control from the application logic, since the decider and the enforcer may be implemented in its middle layer. Some examples

of systems adopting this architecture are Common Object Request Broker Architecture (CORBA), Enterprise JavaBeans (EJB), and JAAS [29].

However, when considering application level access control requirements, these systems present some limitations related to the authorization expressiveness they can support and the type of objects they can protect. In general, these systems do not support more expressive authorizations with domain specific information and they can not protect applications' specific objects, such as specific functionalities.

Supporting the definition and enforcement of more expressive authorizations is performed through the use of predicates. Predicates may be defined with different types of specific information from the application domain, as illustrated in Figure 2.2. Using this specific information in the definition of the predicates, makes it mandatory the decider to have access to that information at the time of the evaluation of the legitimacy of the access.

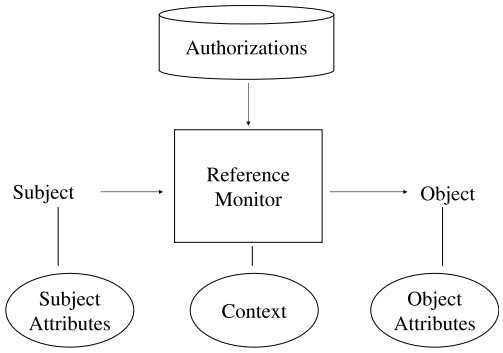

Figure 2.2: Abstract reference monitor extended with authorizations with predicates.

Within more generic solutions, a specific module implements the decider and the invoker supplies the domain specific information to the module at invocation time [38, 36]. Typically, these solutions require that applications perform the invoker function, not avoiding the scattering of access control code over the application code.

Alternatively, the specific module can itself obtain the domain specific information through requests to the applications [8, 15]. This approach allows the decider to be invoked by middleware

systems, promoting the separation between access control and application code.

However, within solutions that implement the decider within a specific module, the domain specific information that is available to the decider is constrained by the application interface predefined.

Additionally, protecting application specific objects constraints the separation between the enforcer and the application logic, resulting in the scattering and in the entanglement of that code over the application modules.

To avoid scattering enforcer code throughout the application, many techniques addressing the problem of scattered concerns have been used, such as design patterns, particularly the proxy pattern [22], meta-level or reflexive architectures [1], and, more recently, AOP [9], in which our work fits.

Though AOP presents potential benefits in the modularization of concerns whose implementation would be naturally scattered through the application code, there are not many examples of its application to real cases in the literature [23, 11]. Bostrom [11] uses this paradigm to add encryption to a healthcare database application. Viega *et al.* [45] and Shah and Hill [39] apply AOP to enforce secure coding practices.

Considering access control, this paradigm can make easy the enforcer implementation. During the development process, the access control code is implemented separately. However, after the composition (the weaving process), this code is an integral part of the application [47].

De Win [47] uses AOP for access control enforcement, analyzing Aspect-Oriented Software Development (AOSD)-based development to the development of a secure File Transfer Protocol (FTP) server.

Bodkin [9] analyzes the requirements of AOP application to security, focusing on the expressiveness of the language constructs that enable the specification of join points.

More recently, Verhanneman et. al [44] proposed a prototype of a modular access control service that can enforce expressive policies, while accounting for application-specific state, without requiring invasive changes to the application, using CaesarJ (an aspect-oriented language) so that

it is possible to change the policy without modifying the application. One of this prototype goals is, like Zás, reuse. Nevertheless, it does not supply none of the advanced features provided by Zás, which will be explained in Chapter 4.

2.3 Access control in Java

The Java virtual machine can run mobile code such as applets [6]. Its initial access control model aims to prevent unauthorized Java applications to access or modify computer resources (for instance: files). This way, within this model, the subject is defined according to the code origin - it is a code-centric model.

JAAS was first introduced as an extension to the J2SE (Java 2 Platform Standard Edition) Development Kit (JDK) and became part of its core with version 1.4. [6]. JAAS added a new access control model to the Java language: a user-centric model. Besides users can be subjects, protected objects can also be specific application resources, i.e., specific application functionalities.

In JAAS, class `Subject` is used to represent a subject (e.g., a user) authenticated in a given system. A subject is an aggregation of principals[3], in which each principal represents one of the different "entities" from whom the subject derives its authority, e.g., a principal may be a username, the name of a group to which she is associated, or a role that she performs. This way, JAAS also supports the RBAC model.

However, the default JAAS does not support runtime changes to the access control policy: its security manager, responsible for the definition of the access control policy, cannot change the principals' authorizations and reflect the changes into the application without restarting it. Additionally, JAAS does not support predicates, which limits the expressiveness of its authorizations. Finally, it has the drawback of not allowing the separation between the enforcement code and the application logic when we try to protect specific protected objects: the invocation of the enforcer

[3] "person [entity] from whom an agent's [subject's] authority derives"[4].

[4] "principal". *Merriam-Webster Online Dictionary*, 2007. http://www.merriam-webster.com/ (The 31st of March, 2007).

must be explicitly made in the application wherever it requires access control, leading to code scattering and to entanglement of concerns.

Laddad [26] modularized the application of JAAS using AOP, eliminating the invasive nature of the enforcement code. Nevertheless, this proposal presents some limitations, as we will detail in Chapter 3.

2.4 Access control in object-oriented databases

Access control models for OO Database Management Systems (DBMS) present, as specific requirements, the incorporation of OO concepts, such as, the difference between classes and class instances, class hierarchies, composed objects and versions of objects. Inheritance and versioning makes access control harder. For example, as pointed out by Strahorn [40], "if a given user has access to a parent class, is there any implicit permission or restriction on access to a child class?".

Consequently, to address these requirements, the access control model defined and supported by ORION OO DBMS[5] presents the following contributions [40]:

1. the identification of the access modes adapted to the systems: read, write, generate and read definition,

2. the identification of the objects that need to be protected, where it is pointed out the separation between the class concept and the protected objects: class and the set of class instances,

3. the definition of the hierarchical relations between access modes and between protected objects. These relations are used on the definition of implication rules of authorizations, which allows us to obtain implicit authorizations from explicitly defined authorizations, according to the following criteria:

   (a) the hierarchical relations pre defined in the model,

[5] "ORION was a prototype OO DBMS developed at MCC, an American consortium by Won Kim and his group.", in http://web.bilkent.edu.tr/Online/oofaq/oo-faq-S-8.12.0.3.html.

(b) authorizations defined for a class are propagated to all subclasses,

(c) authorizations of a complex object are propagated to its sub objects,

(d) authorizations defined for a generic version of an object are propagated to all its versions (the authorizations defined specifically for a version are applied to that version only).

Although a variety of models addressing access control in OO DBMS have been proposed, as pointed out by Strahorn [40], none of them have been of sufficient merit to be completely incorporated into commercial products. As a result, each different commercial next-generation database provides its own flavor of security. In order for these databases to become as accepted in industry as the relation databases, the proposed models and some new ideas need to be combined into a standard for security in next-generation databases.

Though the focus of this work is not answering to questions like “what is the relation between the access control specified on a method of a class whose subclasses override it”, in our opinion, this kind of relations should be explicitly defined in the code since it augments the expressiveness of that code and hence increases the code readability.

2.5 Conclusions

In this chapter, we present the main access control concepts and we overview the main contributions and open problems related to the application of the abstract reference monitor, to avoid the scattering and entanglement of the access control concerns throughout the logic of the application, which constitutes the problem we are addressing in this work. Finally, the chapter discussed the additional problems that controlling accesses in object-orientation brings, such as the relation between a class and its sub classes. Though we present a framework to modularize access control in an OO language, we did not address those problems, since we needed to clearly limit the boundaries of our work.

Chapter 3

A toy application and possible solutions

What do the classical solutions to the access control problem look like? What are their main drawbacks? What exactly are we trying to solve? This brief chapter attempts to hint at the answers to these questions by proposing a very simple toy application and presenting solutions to the access control problem using some of the techniques discussed in the previous chapter, ranging from ad hoc solutions to the use of JAAS improved by the use of AOP and Java 5 annotations.

3.1 Toy application

Suppose we have a simple banking system with an operation – `debit()` – that needs to be access controlled,

```
public class Account {
   private User owner;

   public Account(User owner) {
      this.owner = owner;
   }
   public void debit(float amount) {
      //...
   }
}
```

and the following access control rules,

1. Business operations are for authenticated users only, which are instances of a class `User` representing the subject. Users have their privileges in the system assigned from a set of abstract access modes,

```
public class User {
   private Collection<String> abstractAccessModes;
   private String name;

   public User(String name) {
      this.name = name;
      this.abstractAccessModes = new HashSet<String>();
   }
   public Collection<String> getAbstractAccessModes() {
      return abstractAccessModes;
   }
}
```

   and `debit()` requires abstract access mode `debit`.

The access control policy in this case would be defined by the tuple: $(authenticatedUser, debit, debit(), true)$.

The testing code for this example would be like:

```
public class Test {
   public static User login() {
      // for simplicity, it simply returns a new instance of user
      // with the abstract access mode ‘‘debit’’.
   }
   public static void main(String args[]) {
      User authenticatedUser = Test.login();
      Account account = new Account(new User("John Doe"));
      account.debit(100.0f);
   }
}
```

3.2 Ad hoc

In an OO ad hoc solution, method `debit()` needs an extra argument, the authenticated user, so that it is possible for the method to evaluate the legitimacy of the access. Method `hasAbstractAccessMode()` simply checks if the abstract access mode passed as argument is contained in the authenticated user abstract access modes:

```
public class Account {
   public void debit(float amount, User authenticatedUser) {
      if(!authenticatedUser.hasAbstractAccessMode("debit")) {
         throw new AuthorizationException();
      }
      // ... as before
   }
}
```

it is now necessary to pass the authenticated user to `debit()`,

```
public class Test {
   public static void main(String args[]) {
      // ... as before
      account.debit(100.0f, authenticatedUser);
   }
}
```

This approach forces us to create non-reusable access control mechanisms from scratch. It also leads to the scattering and entanglement of the access control code throughout the application code requiring access control.

3.3 JAAS

Our main focus while studying existing authorization tools was JAAS, since it is a standard for authentication and authorization in Java [42] and an integral part of the JDK.

In JAAS, there is no need for an extra argument in method `debit()`, containing the authenticated user, since JAAS fetches the currently logged in subject through stack inspection. Therefore,

on the callee side – method `debit()` – it would be only necessary to call `checkPermission()`, which associates the abstract access mode `debit` to the method:

```
public class Account {
   public void debit(float amount) {
      AccessController.checkPermission(new BankPermission("debit"));
      // ... as before
   }
}
```

However, on the caller side – method `main()` – it is necessary to wrap the call to method `debit()` inside a `doAsPrivileged()` invocation to which the authenticated subject[1] must be passed as argument, followed by an inline anonymous class extending `PrivilegedAction` and overriding its `run()` method, and finally by an instance of the `AccessControlContext` to be tied to the specified subject and action which, when set to null, forces the instantiation of a new `AccessControlContext` with an empty collection of `ProtectionDomains`.[2]

```
public class Test {
   public static void main(String args[]) {
      // ... authenticate subject and get user advice code

      Subject.doAsPrivileged(authenticatedUser, new
         PrivilegedAction<Test>() {
         public Test run() {
            account.debit(100.0f);
            // it has to return something. In this case, since debit()
            // is not a procedure, it returns null.
            return null;
         }
      }, null);
   }
}
public class User implements Principal {
   // ...
}
```

[1] JAAS needs a special kind of subject, which means that the original authentication solution would need to be slightly changed. However, since this is not our focus, we did not include it here.

[2] See http://java.sun.com/j2se/1.4.2/docs/api/java/security/ProtectionDomain.html.

```
public class BankingPermission extends BasicPermission {
   // ...
}
```

JAAS is a good, non-ad hoc solution to the problem of supplying our toy application with access control, especially in what concerns authentication. In fact, JAAS authentication model is based on the concept of Pluggable Authentication Modules (PAM), allowing developers to easily change from one authentication module to another, by simply changing a policy file, without the need to recompile the application.

However, JAAS is implemented and typically applied using OO approaches, thus being prone to the common problems of code scattering and tangling: as we have seen, code must be added to the application classes in order to implement authorization both at the callee and at the caller code. As stated by Scott [34], JAAS requires considerable configuration effort and is by nature, totally invasive.

Security policy files are used to specify the principals and what are their abstract access modes. E.g., for the current example, we would need a policy file such as:

```
grant Principal banking.User "authenticatedUser" {
   permission test.BankingPermission "debit";
};
```

JAAS access control has some weaknesses but it is still clearly a non-ad hoc solution, meaning that developers do not have to create it from scratch. Its main advantages are that

1. it is used by the access control of the Java language itself,
2. it separates the security policy from the application code, recurring to external files defining the principals and their abstract access modes granted,
3. it is a standard in Java.

However, JAAS access control forces changes both to the caller and to the callee code. JAAS, thus is very intrusive. All calls to access controlled methods must to be wrapped inside

`Subject.doAsPrivileged()` blocks of code, leading to reduce code legibility and increased maintenance efforts. This drawback, combined with the (justifiability) complicated nature of JAAS (and implementation details) makes it hard to adopt the framework in real applications.

3.4 JAAS with AspectJ

Laddad [26] addresses the intrusive nature of JAAS and shows the potentials and strengths of using AOP to modularize access control.

```
public class Account {
   public void debit(float amount) {
      // ... as before
   }
}
public class User implements Principal {
   // ...
}
public class BankingPermission extends BasicPermission {
   // ...
}
```

The original toy project code practically suffers from no changes with the introduction of the access control concerns, since the access control related code is all modularized in the aspect. In Laddad's proposal, there is an abstract aspect that needs to be concretized later in the client application, as will be explained later in Chapter 4. That base aspect has an abstract pointcut to be defined in specific implementations of the aspect and an abstract method returning the JAAS `Permission` class associated with a given method.

```
public abstract aspect AbstractAuthAspect {

   private Subject authenticatedSubject;
   public abstract pointcut authOperations();

   // authenticate subject advice code

   public abstract Permission getPermission(JoinPoint.StaticPart
```

```
        joinPointStaticPart);

    Object around(final Account account) :
        authOperations() && !cflowbelow(authOperations()) {
        Subject.doAsPrivileged(authenticatedSubject,
                        new PrivilegedAction<Object>() {
            public Object run() {
                return proceed(account);
            }
        }, null);
        return null;
    }
    before() : authOperations() {
        AccessController.checkPermission(
            getPermission(thisJoinPointStaticPart);
    }
}
```

In our toy project, the previous aspect would be implemented as follows, leaving the client code totally free from any access control related code.

```
public aspect BankingAuthAspect extends AbstractAuthAspect {
    public pointcut authOperations() :
        execution(public void Account.debit(float));

    public Permission getPermission(JoinPoint.StaticPart
        joinPointStaticPart) {
        return new BankingPermission(joinPointStaticPart.
            getSignature().getName());
    }
}
```

AOP enables to achieve a better modularization and separation of concerns, by putting the code related to authentication and authorization inside the aspect. However, Laddad's proposal assumes that each method has associated to it an abstract access mode having the same name as the method. Therefore, it would be necessary as many abstract access modes as the protected objects in our application. Finally, Laddad's proposal is based on quantification, which is a problem in AOP because of the difficulties in controlling the scope of the application of the aspects.

3.5 JAAS with AspectJ and annotations

This solution uses annotations, allowing to both mark the protected objects and define the abstract access mode required to access that protected object. Therefore, we would create an annotation, as follows,

```
@Retention(RetentionPolicy.RUNTIME) public @interface
AccessControlled {
   public String name();
   public String actions();
   public Class<Permission> permissionClass;
}
```

having as its elements the name and actions needed to pass to the creation of class `Permission` in JAAS, and the JAAS permission class itself, needed to access the protected method. That annotation would then simply be added next to the signature of the method we want to have under access control,

```
public class Account {
   @AccessControlled(name = "debit", actions = "",
      permissionClass = BankingPermission.class)
   public void debit(float amount) {
      // ... as before
   }
}
public class User {
   private Collection<Permission> abstractAccessModes;
   // ...
} public class BankingPermission extends BasicPermission {
   // constructor with a String parameter
}
```

The access controller aspect would be modularized as

```
public abstract aspect AbstractAuthAspect {

   private Subject authenticatedSubject;
   public pointcut authOperations() :
      execution(@AccessControlled * *..*.*(..));
```

```
    // authenticate subject advice code

    public Permission getPermission(AccessControlled ac) {
       final Class permissionClass = ac.permissionClass();
       final String name = ac.name();
       final String actions = ac.actions();
       // ... instantiate the permission with the
       // ... appropriate arguments
       return permission;
    }

    Object around(final Account account) :
       authOperations() && !cflowbelow(authOperations()) {
       Subject.doAsPrivileged(authenticatedSubject,
                        new PrivilegedAction<Object>() {
          public Object run() {
             return proceed(account);
          }
       }, null);
       return null;
    }
    before() : authOperations() {
       AccessController.checkPermission(
          getPermission(thisJoinPointStaticPart);
    }
}
```

Since there is no abstract method in the previous solution, the implementation of the concrete aspect is straightforward, simply being necessary a non-abstract aspect to extend it.

```
public aspect BankingAuthAspect extends AbstractAuthAspect {
}
```

Annotations add more semantics to code thus improving its quality, since it augments its expressiveness while reducing scattering and entanglement. Annotations may be though of as allowing the programmer to express in the code its required semantics, still oblivious of the exact way in which these semantics will actually be implemented.

The use of annotations to mark the access controlled methods is beneficial since, e.g., they force the invocation of `checkXXX()` to be executed at the beginning of the access controlled method and never in the middle or in the end, and addresses the problem of the separation of concerns, since methods are marked as access controlled but do not have any access control-related code in their bodies.

As presented in this example, AOP together with annotations is a powerful technique to modularize access control. However, this solution depends on the use of JAAS, it does not decrease its required configuration effort, and it does not take into account predicates, i.e., access control rules based on domain or context specific information. E.g., adding a new access control rule specifying that only the owner of the account is allowed to debit that account is not easy to accomplish with JAAS. Zás, as will be seen in Chapter 4, solves this.

Chapter 4

The Zás framework

In this chapter we present Zás, an access control framework for Java applications that applies the abstract reference monitor proposed by Anderson. It is reusable and supports authorizations with domain specific information.

4.1 Access control model

Generically, an authorization is defined as a tuple $(s, m, o, pred)$ stating that subject s has the access mode m over the protected object o if the predicate $pred$ is true.

In this section we refine the generic authorization definition in order to meet access control requirements of Java applications.

- *Subjects*: The Zás framework is not restricted to any kind of subject, i.e., subjects can represent users, groups of users or roles, etc.
- *Access modes*: The Zás access control model uses the concept of abstract access mode: abstract access modes are granted to subjects and are associated to operations that can be done over protected objects.
- *Protected objects*: The Zás framework aims to protect fields and methods. The operations

we can perform over fields are sets and gets and the operation we can perform over methods is execute. To reduce the number of access control requirements we need to define, Zás also supports their definition for classes and interfaces. These access control requirements are propagated to all non private members of the class/interface (attributes and methods).

- *Predicates*: Generically, predicates are used to augment the authorizations expressiveness, supporting the definition of more fine-grained authorizations, restricting their application [18]

  Predicates or conditions may use different types of information, such as:

  - user characteristics, e.g., name, date of birth, gender, nationality, etc.
  - object characteristics (the access control depends on the content)
  - external conditions, e.g., the access localization (the access control depends on the context), previously performed accesses (the access control depends on the context flow)
  - relation between entities

This way, we split authorizations into two tuples: $(s, m, pred)$ and (m, op, po). The first tuple defines that subject s has the abstract access mode m if $pred$ is true. The second one defines that any subject that has the abstract access mode m can perform the operation op over the protected object po. From now on, we call the fist tuple as an authorization and the second one as an access control requirement.

4.2 Architecture

The architecture of Zás is based on the abstract reference monitor, as illustrated in Figure 4.1.

The Zás enforcer, the main contribution of this work, is responsible for intercepting every access to protected objects, gathering the context information necessary to the evaluation of au-

thorizations, passing it to the decider (which can be supplied by client code), and finally for enforcing the decision taken.

The Zás decider is responsible for deciding whether a subject can legitimately access a protected object, and for returning its decision to the enforcer, so that it may grant or deny access from the subject to the protected object.

These functions include the following steps:

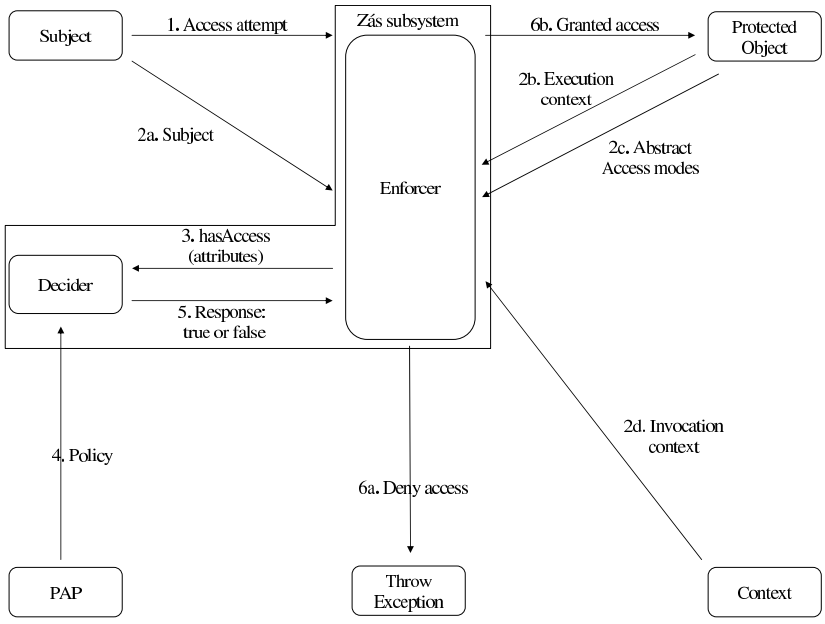

Figure 4.1: The architecture of the Zás access control framework.

1. The subject attempts an access to the protected object. The enforcer intercepts the access.
2. The enforcer collects the context information, particularly the one that the decider needs:
   - 2a. The subject.
   - 2b. The protected object and its usage context, e.g., the object, the method and the arguments associated to the invocation of a method.

2c. The abstract access modes required to access the protected object (specified through Java 5 annotations and Zás eXtensible Markup Language (XML) files, and gathered through reflection) and other architectural meta-information that the enforcer may need.

2d. The static context in which the protected object usage has been requested, e.g., the method containing the code that tries the access.

3. The decider gets the context information required to evaluate the legitimacy of the access.

4. The decider searches in the access control policy defined externally from the code (e.g., in JAAS policy files or XACML).

5. The decider returns to the enforcer the result of its evaluation.

6. The enforcer enforces the decision that was taken:

6a. Either an exception is thrown, in the case of an access denial,

6b. or the access proceeds, otherwise.

4.3 Design

The design of the framework is presented here using extensions to the Unified Modeling Language (UML) [37, 10] proposed by Jacobson and Wei [25], to support aspect orientation, and by Halvorsen and Haugen [24], to represent exceptions being thrown in sequence diagrams. To simplify this presentation, the explanations and the diagrams were applied to the toy application presented in Chapter 3.

Figure 4.2 presents the use case associated to the debit of a bank account. An authenticated user tries to debit an account. To perform that operation, though, she requires authorization. The access control is supplied by the subsystem Zás, which in this case is a realization of the Zás

framework in the form of an access control subsystem adapted to the banking application. This realization consists of:

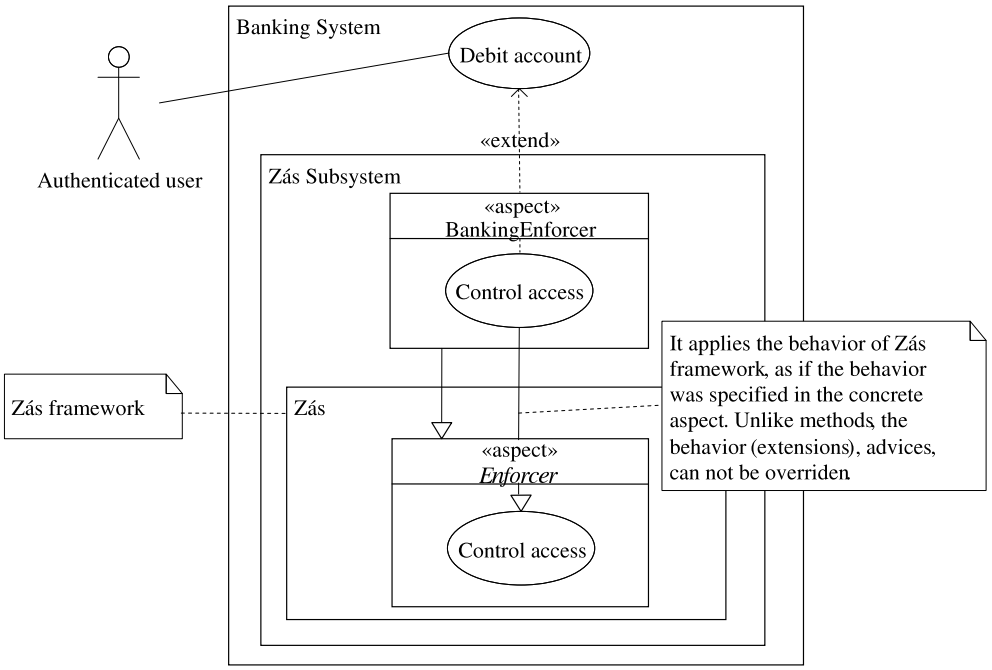

Figure 4.2: “Debit account” use case diagram.

- the definition of the default decider (and other optional deciders specific to the application)
- the specification of the protected objects whose access should be controlled and the abstract access modes associated to them,
- the optional external specification of the access control requirements, and
- the parametrization of the framework’s generic code with the application class representing the subjects of access control.

In the case of the account debit, the access control may be seen as an extension to the basic use case, providing it, in a non-invasive way, with the required access control functionalities.

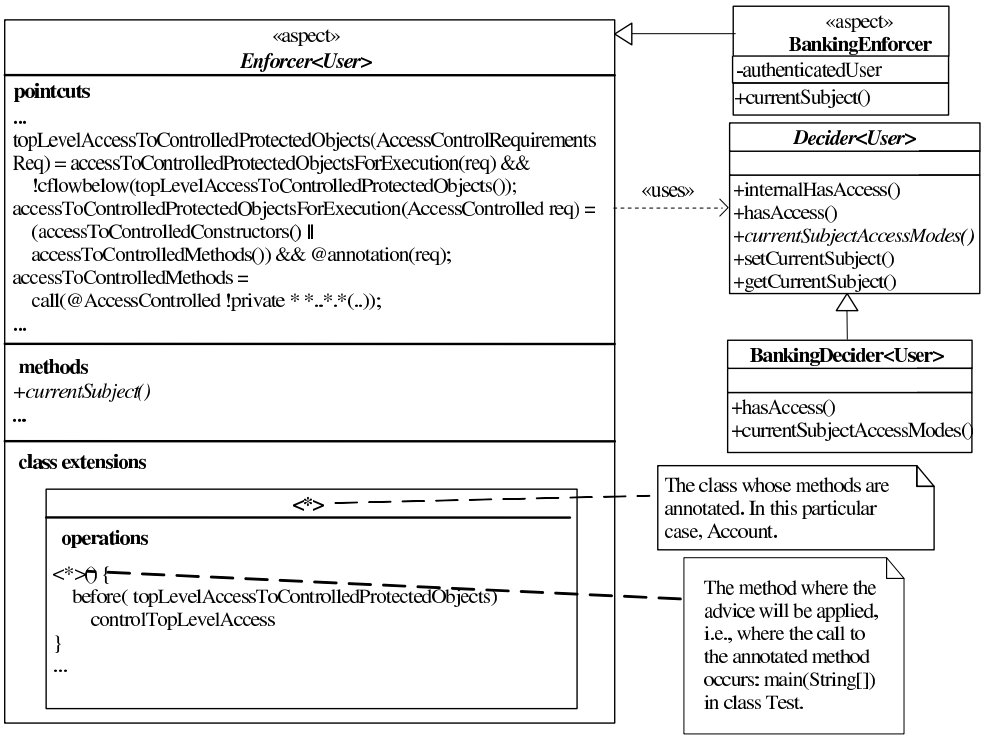

Figure 4.3: Class diagram for the banking system case study.

As can be seen in Figure 4.3, the main entities in Zás, the enforcer and the decider, are modularized respectively by the abstract and generic aspect `Enforcer` and by the abstract class `Decider`. Zás subsystems are realized in each application by the concretization and parametrization of `Enforcer`, by the non-abstract class extension to the decider `Decider`, and by the annotation(s) `AccessControlled` contained or injected in the application code.

The parameter of the generic aspect `Enforcer` is used by client code to specify the class that represents the subjects of access control in the application. In the banking system case, subjects are simply instances of class `User`.

The concretization of the abstract aspect `Enforcer` is essential to make the access control available in the application: in AspectJ, abstract aspects have no effect on their own. This con-

cretization requires the client code to implement the abstract methods of `Decider`, particularly the method `currentSubject()`, which returns the subject involved in the current access attempt and `defaultAccessVerifierClass()`, which defines the default decider class (this class must be a non-abstract extension to class `Decider`, having one no-argument constructor), responsible for evaluating the legitimacy of the access attempt.

Additionally, the client code must extend class `Decider` and implement its abstract method `currentSubjectAccessModes()`, which returns a textual representation of the abstract access modes that the subject has permission to use, which are necessary to evaluate the legitimacy of an access attempt.

The sequence diagram in Figure 4.2 represents the behavior of Zás subsystem in the case of the debit operation. The messages are exchanged between the objects involved in the use case, including its extension supplied by the Zás subsystem. The behavior added by the subsystem to the debit operation occurs before its invocation. This way, the invocation happens only if the authenticated user has access, after the evaluation of the access by the decider in the Zás subsystem (which in this case is the decider module provided by Zás framework).

4.4 Implementation

The generic aspect `Enforcer` has as a parameter `Subject`, of which the access control subjects are supposed to be instances:

```
public abstract aspect Enforcer<Subject> ...
```

The framework may thus be parameterized with the specific type of the subject used by a given application, in this case with class `User`, as will be seen later.

The extension to the behavior of the debit account operation, extension that corresponds to the access control concern, is defined by a `before()` advice from the aspect:

```
// Advice name: controlTopLevelAccess
before(AccessControlled requirements) :
```

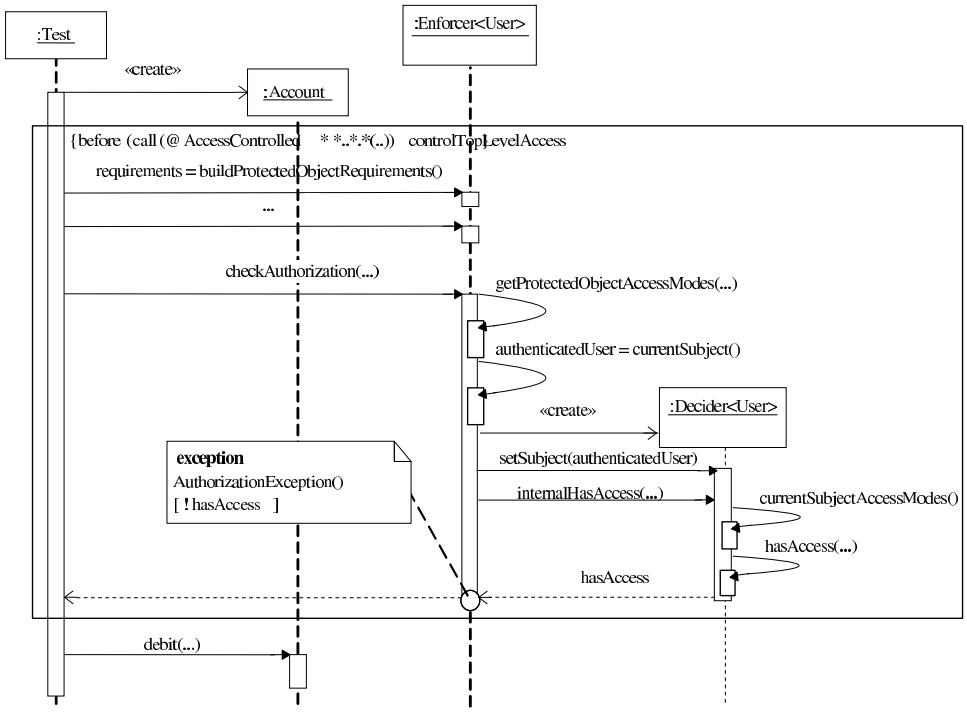

Figure 4.4: “Debit account” case study sequence diagram.

```
   topLevelAccessToControlledProtectedObjects(requirements)
   // ... refinements in the advice definition.
{
   // ...
   checkAuthorization(
      AccessControlRequirements.buildFrom(
         requirements, ProtectedObjectKind.convertToEnum(
            thisJoinPoint.getKind())),
      thisJoinPoint, thisEnclosingJoinPointStaticPart,
      false, false);
   // ...
}
```

This advice is applied to the invocation (call to a method, or set or get to the value of a field) of the protected objects annotated with `AccessControlled` that are not invoked within the control flow of a similarly annotated method (`topLevelAccessToControlledResources`). The method `buildFrom()` returns a new instance of class `AccessControlRequirements` created from the protected object's annotation. That class instance represents the annotation elements' data and is associated to the kind of protected object (method or constructor, and set or get to the value of a field). That class represents both the elements data from the Zás framework access control annotations and the access control requirements specified in the Zás XML files, which will be presented later.

The most important part of the behavior of this advice is implemented in the method `checkAuthorization()`: [1]

```
private void checkAuthorization(
   AccessControlRequirements requirements,
   JoinPoint joinPoint, JoinPoint.StaticPart enclosingStaticPart,
   boolean isShallow, boolean inCflowOfAccessControl) {

   //...
   Subject currentSubject = currentSubject();
   Decider<Subject> decider = deciderClass(requirements,
      joinPoint, !inCflowOfAccessControl);
   String protectedObjectAbstractAccessModes =
      getProtectedObjectAccessModes(requirements, joinPoint);
      decider.setCurrentSubject(currentSubject);
   //...
   if (!decider.internalHasAccess(protectedObjectAbstractAccessModes,
      joinPoint, enclosingStaticPart))
      handleSituationWhenAccessNotGranted(joinPoint.getStaticPart());
}
```

The parameter `joinPoint` in the invocation of method `deciderClass()` is required so that it is possible to register the decider and associate it to that join point (there is only one

[1] It is usually a good practice to define the arguments passed to a method as final. However, in this thesis, for presentation purposes and due to the limitations in terms of the page width, we decided not to include them.

instance of decider per protected object, which is created the first time that the protected object is accessed and reused later on). The argument `isShallow` control the depth of the access control enforcement, as will be further explained in Section 4.6.2.3. The argument `inCflowOfAccessControl` controls either the decider class should be registered for the current join point or not: the registration is performed only in top level accesses.

The method `checkAuthorization()` obtains the subject through the invocation of the abstract method `currentSubject()`, implemented in the concrete aspect `BankingEnforcer`, and obtains the protected object's abstract access modes from the information specified in its meta-information (e.g., in annotation `AccessControlled`, presented later in this section), by invoking `getProtectedObjectAccessModes()`.

In the method `checkAuthorization()` it is also necessary to obtain an instance of the decider class specified in the protected object's meta-information and invoke its method `internalhasAccess()`, which compares the abstract access modes expression defined in the protected object's access control requirements with the abstract access modes the subject has permission to use, obtained through the invocation of the abstract method `currentSubjectAccessModes()` (implemented in the concrete decider (`BankingDecider`)), returning true only if the subject has the required abstract access modes to access the protected object. Additionally, it invokes method `hasAccess()`, optionally implemented in client code, thus enabling to perform additional access control verifications.

```
@Privileged
public class Decider<Subject> {
   public final boolean internalHasAccess(
      String protectedObjectAccessModes,
      JoinPoint joinpoint,
      JoinPoint.StaticPart enclosingStaticPart) {
      boolean result = false;
      // ...
      return result &&
         hasAccess(protectedObjectAccessModes, joinpoint,
         enclosingStaticPart);
   }
```

```
    public boolean hasAccess(String protectedObjectAccessModes,
       JoinPoint joinpoint, JoinPoint.StaticPart enclosingStaticPart) {
       return true;
    }
}
```

When `internalHasAccess()` returns true, the access to the protected object proceeds immediately. When it returns false, `checkAuthorization()` throws a runtime exception. Notice that it is a runtime, unchecked exception, as in JAAS, therefore making it possible for Zás to be totally non-invasive, since unchecked exceptions do not have to be declared as part of the methods signatures.

Apart from the arguments already mentioned, `hasAccess()` gets also the execution (dynamic) and invocation (static) contexts of the access attempt.

The *dynamic* context is necessary so that the access control decision may be based on data. However, if the impact in the application performance is deemed too large, a future version of the framework will possibly define two kinds of authorizations, as suggested by Laddad [28]: `AccessControlled` and `DataDrivenAccessControlled`, together with two base deciders – one receiving dynamic information, the other static.

It is important to notice that the class `Decider` and all its subclasses[2] are marked as privileged (further details are presented in Section 4.6.2.4).

This is necessary because these classes often need to obtain access to protected objects, particularly when it is necessary to compute the value of predicates using domain specific information, e.g., protected object's properties. This may be a problem, since in the context of these classes, one may access every data of a given application, therefore, this is an open issue for further investigation.

In AspectJ, pointcuts may be either named or anonymous. In general, named pointcut definitions may be overridden by definitions with the same name in derived aspects. When it is not

[2] Annotation `@Privileged` is annotated with `@Inherited`.

desirable, pointcuts may be qualified as final in the base aspect or even private, when they are seen as implementation details.

The two main pointcuts from aspect `Enforcer` are private, i.e., they are implementation details and hence they cannot be overridden:

```
private pointcut topLevelAccessToControlledProtectedObjects(
   AccessControlled requirements) :
   accessToControlledProtectedObjectsForExecution(requirements) &&
   !cflowbelow(
      accessToControlledProtectedObjectsForExecution(AccessControlled));
private pointcut accessToControlledProtectedObjectsForExecution(
   AccessControlled requirements) :
   (accessToControlledConstructors() ||
      accessToControlledMethods()) &&
   @annotation(requirements);
public pointcut accessToControlledMethods() :
   call(@AccessControlled !private * *..*.*(..));
```

This is due to the fact that any change in these pointcuts may compromise the Zás subsystem.

On the contrary, derived concrete aspects of `Enforcer` may override pointcut `accessToControlledMethods`. For instance, aspect `BankingAccessController` could refine the pointcut definition in such a way that the private methods would also be subject to access control.

By default, and since this framework is still a prototype, the used join points correspond to invocations of methods annotated with `AccessControlled`: the application of advices to these join points has the advantage of allowing one to obtain not only the dynamic context of execution, but also the static context of invocation.[3]

Nevertheless, in an implementation of the framework for production, this choice is not the most appropriate, for security reasons. Advices, in the case of invocations, are weaved in the code that accesses the protected object. If this code is external to the application or, for some reason, can not be manipulated by the AspectJ compiler, the weaving of the access control advices may be

[3]We have already started an implementation using execution instead of call, though, in the branch "B_zas-new-annotations", which may be found in the Web site of the project at http://sourceforge.net/projects/zas/.

impossible, compromising the security of the application. Therefore, the Zás framework should, by default, control the executions, since the compiler has full access to the application of which the subsystem Zás is a part.[4] Additionally, the protected object as defined in Zás is the method itself, i.e., the execution of the method and not its invocation.

The control of method executions, not invocations, will make it harder to implement some of the current features of the Zás framework, described in Section 4.6.2: it will no longer be possible to obtain the static context of invocation through the implicit variable `thisEnclosingJoinPointStaticPart`, thus, the implementation of some current features from Zás will be harder. Implementing execution side access control might require the observation and the parsing of the Java stack trace. However, the Java stack trace does not provide the full signature of the methods on the stack, which means it is not possible to access the method by simply parsing the stack. The solution for this problem would include the creation of a pseudo-stack, managed by Zás, which works as an add-on to the Java stack trace, in order to augment its information. This implementation has already begun, as detailed in Section 4.6.3.

The annotation `AccessControlled` is used by the Zás framework to specify the meta-information used to control the access to methods:

```
public @interface AccessControlled {
   String value() default "#";
   Class<? extends Decider> deciderClass()
      default Decider.class;
   Depth depth() default Depth.DEEP;
   Class[] trusts() default {};
   boolean inherited() default false;
   boolean suspicious() default false;
}
```

The `value()` element is used to declare the names of the abstract access modes required to access to the protected objects. By default, its value is `''#''`, which represents conventionally the full signature of the protected object (though without return type).[5]

[4] Bug submitted in: http://sourceforge.net/tracker/?group_id=195589&atid=954049.

[5] Using full signatures prevents the collision of the names of the abstract access modes.

The `deciderClass()` element is used to explicitly specify the class responsible for deciding whether the access should be granted. By default, it is the class `Decider`. Notice that different protected objects may define different deciders.

The `depth()` element specifies the enforcement depth, i.e., it specifies whether control access should be performed (or not) during the flow of control of a method that has already been access access controlled. When a method `foo()`, say, has depth `SHALLOW`, it is enough for the subject to have access to the invocation of `foo()`, since the subject will be granted access to any protected object, provided the access occurs within the flow of control of that invocation of `foo()`. By default, the value of this element is `DEEP`, since it is the most restrictive behavior and, as such, the most secure, forcing all accesses to be controlled, at any level of the stack trace.

The `trusts()` element is, by default, empty. It is used to indicate the classes that have automatic access to the protected object. It is used to explicitly declare that the protected object trusts in those classes to access the protected object. This is somehow related to "exporting features" in Eiffel [31], because in Eiffel we may restrict access of a method to a classes, similar to what we may do in Zás with this element. When this element is used, the current subject is not only the entity that has initiated the access attempt but also the classes in which the protected object trusts.

The `inherited()` element controls the inheritance of the access control requirements from the type (class or interface) where the protected object was defined. By default, its value is false, meaning that by default protected objects do not inherit the access control requirements from their type.

Finally, `suspicious()`, when true, forces the access to the protected object to be performed at any level of the stack trace, even when the access depth verification was specified as `SHALLOW` at some lower point in the stack.

From a strict security point of view, all "resources" should be seen as protected objects: it is safer for the application to throw an authorization exception for all access attempts than to implicitly grant access from all subjects to that "resource". However, that would forbid programmers to

restrict the access control scope, for example, to the business layer of the application. Ideally, it should be possible to define an abstract pointcut `scopeForProtectedObjects()`

```
// In aspect Enforcer:
public abstract pointcut scopeForProtectedObjects();
declare @method : scopeForProtectedObjects() :
   @AccessControlled("*");
// ...
```

forcing this way the client code to define it appropriately to her application. The `''*''` is a wildcard that matches for any abstract access mode, i.e., the previous code means that all protected objects require, at least, one abstract access mode.

However, the AspectJ language forces the usage of type patterns in Inter-Type Declaration (ITD), not allowing the usage of pointcuts. Therefore, the generalized access control in the business layer is considered as a best practice of Zás, but it cannot be forced by it.

4.5 Usage

In this section we will present the Zás-based solution to the implementation of the toy project from Chapter 3 with an additional access control rule, stating that only the account owner is allowed to debit the account if she was granted permission to use the abstract access mode `debit`.

The first step in a Zás-based implementation of access control in some application is to identify the protected objects whose access should be controlled. There are two different ways to do this. One of them is invasive, consisting in annotating directly the protected objects whose access should be controlled:

```
public class Account {
   @AccessControlled(value = "debit",
      deciderClass = BankingDecider.class)
   public void debit(float amount) {
      // ...
   }
}
```

```
public class Test {
   // ... as before
}
```

The other is non-invasive. It involves using ITD to inject the necessary annotations in the access controlled protected objects. The drawback of this approach, however, is that readability becomes harder: the access control requirements of a protected object are not specified close to its definition. The Integrated Development Environment (IDE) we used was Eclipse[6], which helps to solve this problem since the protected object is shown with a note indicating that it is being annotated by some aspect. Nevertheless, it is simply a note and it makes programmers depend on the IDE:

```
// In aspect Enforcer:
declare @method : Account.debit(float) :
   @AccessControlled(value = "debit", deciderClass = BankingDecider.class);
```

As a second step, it is necessary on the one hand to configure the security aspects from Zás though the parametrization, extension, and concretization of aspect `Enforcer`, which involves the implementation of the abstract methods `currentSubject()` and `defaultDeciderClass()`, and, on the other hand, at least for this particular case, the definition of an advice that guarantees user authentication:

```
public aspect BankingEnforcer extends Enforcer<User> {
   public User currentSubject() {
      return SecurityContext.getCurrentSubject();
   }
   public Class<BankingDecider> defaultDeciderClass() {
      return BankingDecider.class;
   }
   after() returning(User authenticatedUser) :
      call(public static User Test.login()) {
      SecurityContext.setCurrentSubject(authenticatedUser);
   }
}
```

The advice shown previously executes after any successful login attempt in the application.

[6] Eclipse `http://eclipse.org/` is the most used IDE to program using AspectJ.

Its implementation uses a helper class which is distributed with the Zás framework, `SecurityContext`,

```
public class SecurityContext {

   private static InheritableThreadLocal currentThread =
      new InheritableThreadLocal();

   private SecurityContext() {
   }

   public static <Subject> Subject getCurrentSubject() {
      return (Subject) currentThread.get();
   }

   public static <Subject> void setCurrentSubject(Subject currentSubject) {
      currentThread.set(currentSubject);
   }
}
```

This class has two methods, `setCurrentSubject()`, which associates a given subject with the current thread, and `getCurrentSubject()` which returns the subject associated to the current thread.

The last step is to define a class representing the decider and extending `Decider`. This class may make use of information which is specific in the application. The class `BankingDecider` grants access to the debit operation exclusively to the owner of the account if she was granted permission to use the abstract access mode `debit`, as specified in the annotation `AccessControlled` of method `debit()`:

```
public aspect BankingDecider extends Decider<User> {
   public Collection<String> currentSubjectAccessModes() {
      return getCurrentSubject() == null ? new HashSet<String>() ?
         getCurrentSubject().getAbstractAccessModes();
   }
   @Override
   public boolean hasAccess(String protectedObjectAccessModes,
      JoinPoint joinpoint, JoinPoint.StaticPart enclosingStaticPart) {
```

```
        Account account = (Account) joinpoint.getTarget();
        return getCurrentSubject().equals(account.getOwner());
    }
}
```

The previous example[7], using AOP, addresses the problem of code scattering and tangling by allowing it to modularize the crosscutting concerns into the aspect, fully separated from the functional code. The changes in existing code before authorization were none. The methods were not changed in order to pass the current logged in user to calculate the user abstract access modes and grant or deny access to protected objects based on the user instance. Zás enforcer is able to get the required access control requirements (the subject, the protected object, and the context of execution) and pass it to the appropriate decider module (e.g., `BankingVerifier`). Additionally, Zás enables the specification of a more dynamic and fine grained security policy based on context information, without increasing the code complexity.

4.6 Features

Since Zás was meant to be a Java/AspectJ framework of classes and aspects for use in Java applications, we have implemented it to address a set of requirements which are detailed next. These requirements were born to support the separation of security concerns from the domain decomposition of the application.

Access control languages such as Ponder [16] or XACML [35] were analyzed to drive the process of deciding the minimum set of requirements that Zás should support. Proposals like Laddad's [26] or Bertino's [7], as well as our own insight gained from the early application of Zás to small projects, were also important sources of motivation and analysis to further refine the requirements. Throughout this chapter, for our code examples, we assume that there is a subject currently logged in, having the abstract access mode `foo` and not having `bar`.

[7] The full code may be seen in `http://zas.cvs.sourceforge.net/zas/banking-toy-project/`.

4.6.1 Basic

As Laddad put it in [27], “the new Java metadata facility, a part of Java 2 Platform Standard Edition (J2SE) 5.0, is perhaps the most significant addition to the Java language to date. By providing a standard way to attach additional data to program elements, the metadata facility has the potential to simplify and improve many areas of application development, including configuration management, framework implementation, and code generation.” The combined power of Java 5 annotations and AspectJ is leveraged by Zás, making it a very expressive access control framework.

Annotations in Java are a powerful way to add more semantics to the Java “resources” (methods, fields, etc.). The programmer simply adds the appropriate meta-information in her programs, i.e., the additional semantics. Those semantics is then taken into account by the Zás framework to enforce an access control policy.

The next snippet of code explicitly states that access to method `foo()`, i.e., the invocation of the method, is restricted to subjects having the abstract access mode `foo`.

```
public class MyClass {
   @AccessControlled("foo")
   public void foo() {}
}
```

When not specified in the annotation, the abstract access mode expression corresponds to a single abstract access mode whose name is the signature of the method, without the return type[8]. Hence, the abstract access mode required to call `foo()` as defined in

```
package mypackage;
public class MyClass {
   @AccessControlled
   public void foo() {}
}
```

is `mypackage.MyClass.foo()`.

[8]The default abstract access mode expression is defined as “#”, corresponding to the protected object signature

With AspectJ, there is a clear distinction between setting and getting the value of fields. There is not, however, the similar distinction between the call of methods that change the implicit object's state and methods that do not change it. Bearing this in mind, only three different annotations have been defined for specifying access control: one for methods, presented before, and two for the setting and getting of the value of fields.

```
package mypackage;

public class MyClass {
   @AccessControlledForQuerying("readBar")
   @AccessControlledForModifying("modifyBar")
   protected int bar;
}
```

The previous example restricts reading access to field `bar` to subjects possessing the abstract access mode `readBar` and the abstract access mode `modifyBar` to change the field's value.

By default, Zás enforces access control only for non-private protected objects. The rationale for this is that private "resources" are usually implementation details, and not accessible from outside the class where they are defined (bearing in mind that, usually, it is a good practice to make fields private resources of the class, the default behavior for controlling the access to fields is limited). It is possible, however, to change the default behavior so that are also controlled private "resources". E.g.:

```
public aspect MyEnforcer extends Enforcer<User> {
   public pointcut accessToControlledMethods() :
      Enforcer<User>.accessToControlledMethods() ||
      call(@AccessControlled private * *(..));
   public pointcut accessToControlledFieldsSets() :
      Enforcer<User>.accessToControlledFieldsSets() ||
      set(@AccessControlledForModifying private * *..*.*);
```

The lack of the required abstract access modes results in throwing an authorization exception. Hence, the following snippet of code results in throwing an `AuthorizationException` exception:

```
public class A {
   @AccessControlled("bar")
   public A() {}
}
```

Zás defines two abstract access modes with a special meaning: `true`, and `false`, whose semantics are, granting access to any subject, and do not grant access to any subject, respectively.

Zás saves internally the relationship between the instances of the decider classes and the corresponding protected objects. This improves the application efficiency, because instantiations occur a single time for each protected object. These instances are then reused each time an access to the corresponding protected object is attempted.

4.6.1.1 Boolean expressions and wildcards

It is possible to compose abstract access modes using Boolean expressions, both statically in-code, and in Zás XML policy files (see Section 4.6.2.2). For instance, in

```
@AccessControlled("foo || !bar")
public void foo() {}
```

the abstract access mode expression requires any subject calling `foo()` either to have abstract access mode `foo` or to lack `bar`.

Currently, Zás supports operators `||` (or), `&&` (and), and `!` (not), as well as the use of parentheses to control evaluation order. Operators `==` (equivalent), and `!=` (not equivalent) will be added soon.

Using Boolean expressions, the abstract access modes of a method may be easily defined as the conjunction of the simpler methods called in its implementation. E.g., a `transfer()` method on the banking system can be defined as a `debit()` on a source account and a `credit()` on a destination account. These methods may be access controlled, and require the abstract access modes `debit` and `credit`, respectively:

```
@AccessControlled("debit && credit")
public void transfer(Account from, Account to, float amount) {
```

```
      from.debit(amount);
      to.credit(amount);
   }
   @AccessControlled("debit")
   public void debit(float amount) {
      this.balance -= amount;
   }
   @AccessControlled("credit")
   public void credit(float amount) {
      this.balance += amount;
   }
```

The operator `||` is useful in all situations where different abstract access modes may grant access to the protected object, while the operator `!` may be used to define the forbidden abstract access modes, i.e., the abstract access modes the subject cannot possess to be able to have access to a protected object. E.g., in an RBAC model, an abstract access mode may reflect the textual representation of the roles associated to a subject. Hence, the following code declares that a subject having the role `EMPLOYEE` or `MANAGER`, with no role `CUSTOMER`, may access the access controlled constructor `Account()`:

```
@AccessControlled("(EMPLOYEE || MANAGER) && !CUSTOMER")
public Account(User owner, float initialBalance) {
   ...
}
```

Regular expressions [33, 41] are also possible for the composition of the abstract access modes. Table 4.1 shows the currently supported wildcards.

Symbol	Meaning
*	0 or more characters
+	1 or more characters
?	0 or 1 characters

Table 4.1: Wildcards currently supported by Zás.

E.g., using

```
@AccessControlled("abstr*") public void foo() {}
```

any call to `foo()` requires a subject having at least one abstract access mode whose name starts with `abstr` (e.g., `abstr` or `abstract access mode`). The use of regular expressions in the declaration of the required abstract access modes results in a disjunction. E.g., if for the previous code there were three different abstract access modes declared as `abstr1`, `abstr2` and `abstr3`, its access control requirements could be similarly defined as:

```
@AccessControlled("abstr1 || abstr2 || abstr3")
public void foo() {}
```

Wildcards can also be used when dynamically composing abstract access modes, of course. In this case, however, they can also be used to specify multiple protected objects in a single step, greatly reducing the number of access control requirements one needs to specify, as shown in the last example of Section 4.6.2.2.

New wildcards could be added, as well. Bearing in mind that a programmer using Zás is familiarized with AspectJ, the wildcards semantics should be the ones in AspectJ [4]. Hence, `..` should be possible in Zás, as suggested by Laddad [28].

4.6.2 Advanced

This section presents advanced, though useful features from Zás. All these features may be seen as optional since they are not required to enable Zás' access control capabilities. Nevertheless, they are often useful, e.g., to control the depth of the access control enforcement.

4.6.2.1 Propagation of requirements

Zás provides a mechanism allowing access control requirements to be propagated from classes to the corresponding non-private methods and fields, which for Zás are the only controlled resources. For instance, the access control requirements of a class may be inherited by all its non-private "resources":

```
@AccessControlled(value = "foo", depth = Depth.SHALLOW)
public class MyClass {
   public void foo() {}
}
```

In this case, `foo()` and the implicitly defined class constructor inherit the access control requirements from their class, i.e., calling the method `foo()` and the constructor `MyClass()` require abstract access mode `foo` and the access control is shallowly enforced.

Similarly, it is possible to propagate access control requirements through non-private fields or to inherit the requirements specified in the Zás' protected objects types. E.g., the access control requirements defined in `MyClass` are propagated to the field `myInt`, since it has no access control annotations. This means that a read access attempt to that field requires the subject to have the abstract access mode `forGet`, and shallow access control. When a write access attempt is performed, the subject requires the abstract access mode `forSet`.

The field `myOtherInt` inherits the access control requirements from its class, namely the depth of enforcement, but is overrides the abstract access mode required for the access, from `forGet` to `newForGet`.

Finally, the field `myAnotherInt` does not inherit the access control requirements from its type, since the element `inherited` is set to false.

```
@AccessControlledForQuerying(value = "forGet", depth = Deph.SHALLOW)
@AccessControlledForModifying("forSet")
public class MyClass {
   int myInt;
   @AccessControlledForQuerying(value = "newForGet", inherited = true)
   int myOtherInt;
   @AccessControlledForQuerying(value = "*", inherited = false)
   int myAnotherInt;
}
```

Notice that there are two different effects in propagation. The first one is static, and leads to all non-private members of a protected object, with the exception of those marked with annotation `@NotAccessControlled`, to also be access controlled. The second one is dynamic, and leads to

all non-private members of a protected object *that have not been explicitly marked as being either access controlled or not access controlled* to dynamically inherit the access control requirements from their enclosing types (see Section 4.6.2.2). Hence, in

```
@AccessControlled(value = "foo", depth = Depth.SHALLOW)
public class MyClass {
   public void foo() {}
   @AccessControlled
   public void bar() {}
   @NotAccessControlled
   public void baz() {}
}
```

`foo()` inherits its access control requirements from class `MyClass`: the abstract access mode name `foo` and shallow access control. However, `bar()`, while access controlled, does not inherit required abstract access modes from `MyClass` because the element `inherited` is, by default, set to false, and `baz()` remains free of any access control.

It is not be possible to propagate access control requirements to types marked as non-access controlled. This happens in the situation where a type is explicitly annotated with `@NotAccessControlled`.

One issue Zás does not deal with so far is the inheritance for packages from the source code, because AspectJ [4] does not allow the capture of package annotations and also because of the limited nature of packages in Java, as detailed in Section 4.6.2.6.

The propagation and/or inheritance of requirements is very useful because it decreases the number of access control requirements one needs to specify in client applications. This feature allows the programmer to mark only the type, causing all non-private protected objects (methods and fields) to also be access controlled, allowing developers to say that, e.g., all accesses to type T are restricted to users having abstract access mode AAM.

Notice that the same occurs when marking protected objects as non access controlled.

```
@NotAccessControlled
public class MyClass {
   public void foo() {}
```

```
    @AccessControlled
    public void bar() {}
}
public class MyExtendedClass extends MyClass {
    public void baz() {}
}
```

The previous results in methods `foo()` and `baz()` not to be access controlled.

Annotations

1. `@AccessControlled`,
2. `@AccessControlledForModifying`,
3. `@AccessControlledForQuerying`, and
4. `@NotAccessControlled`

are defined as `@Inherited`. Hence, they are inherited by subtypes of types annotated with them. This means that methods and fields on subtypes of types with that annotation are subjected to the same rules for the propagation of requirements.

4.6.2.2 Dynamic access control requirements

As indicated in the access control requirements' annotations, the access control requirements are simply initial requirements, which may be changeable at runtime. That is, access control requirements are dynamic. The access control requirements may be set only in the code, or both in the code and in XML. This is configured in client code, through the specification of the access requirements mode. E.g., in the following piece of code, the access control requirements could be specified in the code only:

```
public aspect MyEnforcer extends Enforcer<User> {
    public MyEnforcer() {
        setSpecificationMode(AccessRequirementsSpecificationMode.IN_CODE);
    }
}
```

The possible modes are `IN_CODE`, i.e., specified in Java code only, and `IN_CODE_AND_FILE`, i.e., specified in Java and XML code (the requirements in XML override the ones in the Java code). The default is `IN_CODE_AND_FILE`:

```
public enum AccessRequirementsSpecificationMode {
   IN_CODE, IN_CODE_AND_FILE;
}
```

In code, the requirements are specified using the previously presented access control annotations. In file, there is one XML policy file in Zás, with a known syntax and semantics that may be used to specify the access control requirements for the "resources" that were marked, through the access control annotations, as protected objects. The file `zas-permissions-schema.xsd` in the Zás project code defines the right syntax for the Zás XML policy file. E.g.,

```
package mypackage;

class MyClass {
  @AccessControlled
  public void foo(String s) {}
}
```

specifies that `foo()` is access controlled and initially requires abstract access mode `mypackage.MyClass.foo(String)`. It is possible to change the required abstract access mode using a Zás XML policy file such as:

```
<?xml version="1.0" encoding="UTF-8"?>
<zas xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="zas-permissions-schema.xsd">
   <protected-object name="mypackage.MyClass.foo(String)" type="method">
      <abstract-access-modes>foo</abstract-access-modes>
   </resource>
</zas>
```

In this case, after loading the access control policy file, the required abstract access mode for calling `foo()` is no longer `mypackage.MyClass.foo(String)`, but `foo`.

This adds a lot of flexibility because it is possible to change the access control requirements for a protected object at runtime, without the need to recompile the application, since all one needs to do is to change the Zás XML policy file.

The use of wildcards increases these capabilities, since it is possible to specify the access control requirements at the appropriate granularity level. E.g.,

```
<?xml version="1.0" encoding="UTF-8"?>
<zas xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="zas-permissions-schema.xsd">
   <protected-object name="mypackage.MyClass.foo()" type="method">
      <abstract-access-modes>foo</abstract-access-modes>
   </protected-object>
   <protected-object name="mypackage.MyClass.*" type="method">
      <abstract-access-modes>bar</abstract-access-modes>
   </protected-object>
   <!-- When the attribute type is not specified, it uses method
        as default -->
   <protected-object name="mypackage.*()">
      <abstract-access-modes>baz</abstract-access-modes>
   </protected-object>
</zas>
```

which may be found in a Zás XML policy file, states that all calls to access controlled methods without parameters within package `mypackage` will require abstract access mode `baz`, with the exception of those within class `MyClass`, which require abstract access mode `bar`. Again, method `MyClass.foo()` is an exception, since it requires abstract access mode `foo`. The order is important because Zás will always look for the first occurrence of a matching signature and load the abstract access mode specification. This simplifies the algorithm that loads the access control requirements. Additionally, it simplifies the specification of the requirements and hence its readability.

With the previous access control requirements, the following code would require a subject possessing abstract access mode `foo` to be granted access to method `MyClass.foo()`, abstract access mode `bar` for method `MyClass.bar()` and abstract access mode `baz` for method `My-`

`Class2.baz()`.

```
package mypackage;

@AccessControlled
public class MyClass {
   public void foo() {}
   public void bar() {}
}

package mypackage;

public class MyClass2 {
   @AccessControlled
   public void baz() {}
}
```

Using Zás XML policy files makes it possible to specify the access control rules in two different phases: the first, where the security engineer simply marks the protected objects that are to be access controlled; the second, where she specifies their access control requirements.

4.6.2.3 Depth of access control

By default, the access control is applied for all accesses to protected objects, regardless of the context. Regardless, namely, of the controls which have already been performed in lower levels of the current call stack. This is usually the safest option and thus the most desirable default. However, it may be occasionally necessary to turn off access control in the flow of control of a given method execution.

Element `depth` of the `@AccessControlled` annotation represents the depth of access control. In a way that is reminiscent of copy depth, access control is applied to method execution either in a `Depth.SHALLOW` or in a `Depth.DEEP` manner, depending on the value of this element. Shallow access control means that if access to a method is granted to a subject, it will also be granted to its complete flow of control, effectively *turning off* access control during its execution. On the contrary, if access to a method specifying deep access control is granted to a subject,

which is the (safe) default, it will *not* be automatically granted to all other accesses to protected objects in that method's flow of control.

For example, in

```
public class A {
   @AccessControlled("foo")
   public void foo() {
     new B().bar();
   }
}
public class B {
   @AccessControlled("bar")
   public void bar() {
   }
}
```

A call to `A.foo()` will be possible only if the subject has both the abstract access modes `foo` and `bar` because `A.foo()`, which invokes method `bar()`, requires `foo`, and `B.bar()`, on its hand, requires `bar`. Changing the depth to `Depth.SHALLOW` in the annotation of `foo()`, the access control will *not* be applied during the execution of `foo()`, which means the subject may invoke `bar()` through `foo()`:[9]

```
@AccessControlled(depth = Depth.SHALLOW)
public void foo() {
   new B().bar();
}
```

Using shallow access control is generally considered dangerous. Hence, a mechanism may be devised to short-circuit the consequences of shallow access control. If a given method declares itself to be suspicious, its access control requirements are *not* turned off in the flow of control of a method with shallow access control. For instance, in

```
public class A {
   @AccessControlled(value = "foo", depth = Depth.SHALLOW)
   public void foo() {
      new B().bar();
```

[9] Notice, however, that calling `bar()` directly would still require the abstract access mode `bar`.

```
        new C().baz();
    }
}
public class B {
    @AccessControlled("bar")
    public void bar() {
        new C().baz();
    }
}
public class C {
    @AccessControlled(value = "baz", suspicious = true)
    public void baz() {
    }
}
```

A call to `A.foo()` will fail unless the subject has the abstract access mode `baz`: it is not sufficient for her to have the abstract access mode `foo`, since `C.baz()` is suspicious. On the other hand, the abstract access mode `bar` is not necessary when the call is performed in the flow of control of `A.foo()`, since `B.bar()` is unsuspecting and `A.foo()`'s access control is shallow.

4.6.2.4 Bypasses

Zás provides two mechanisms to bypass access control. The first is more dangerous, and should be used with care: methods may be annotated as privileged, i.e., as turning off all access control within their flow of control:

```
@Privileged public void foo() {
    // ...
}
```

The difference between calling a privileged method and calling a method with shallow access control is that a call to a privileged method always succeeds, while the success of a call to a method with shallow access control depends solely on the current subject having abstract access mode to make the call (except when "suspicious" is used). Privileged, though very dangerous in security terms, can be sometimes useful and even essential. The Zás framework uses them, e.g.,

in the `Decider` class and its subclasses, and thus grant them full access to any protected object. Therefore, the deciders may access domain specific information, which may be access controlled, when the legitimacy of an access attempt requires that context information. However, this may compromise the security of the application, since it may open security breaches and hence, further study is required to control what the decider classes may and may not do.

The second bypassing mechanism, trust, is more disciplined and less dangerous. Instead of marking methods as bypassing access control during their flow of control, regardless of the access control requirements of the intervening protected objects, trust in specified classes is explicitly acknowledged by the callee protected object. For example, given

```
public class A {
   @AccessControlled("foo")
   public void foo() {
      new B().bar();
   }
}
public class B {
   @AccessControlled("bar", trusts = { A.class } )
   public void bar() {
   }
}
```

calls to `A.foo()` require a subject with the abstract access mode `foo`, as usual. However, the call to `B.bar()` from within `A.foo()` will always succeed, since `B.bar()` trusts class `A`. Notice, however, that calls from within the control flow of `B.bar()` will in general be access controlled, since trust does not propagate. This improves even further the safety of trust relationships.

4.6.2.5 Invokers

According to Bertino [7] and Thomas [43], each protected object should be able to specify the allowed set of invokers. The idea is that each protected object should be able to specify the "entities" that may access them. In the Zás framework, we mapped *entities* as the methods that invoke a protected object on someone (an authenticated subject) or something (an authenticated

system) behalf.

This is similar to trusts, though there are important differences. Conceptually, the set of invokers is a way to define a workflow, i.e., to explicitly state what methods are allowed to access one protected object and thus restricting its access to a very strict set of methods. This is accomplished through the explicit specification of the methods that are granted access to a protected object: methods which are not included in the set of allowed invokers are not granted access to that protected object (if the resource declares an empty set, that means no one is able to access it). However, for trust, an object not belonging to the set of trusted classes can still be granted access, depending on the subject's abstract access modes.

They are also different in implementation terms: for trust, the protected object declares its "trust" on objects from a *class*; for the invokers, the protected object declares the *methods* that may legitimately access it.

According to Arnold et al. [3], the elements of an annotation type are constrained by strict rules:

1. The type of an element must either be a primitive type, a `String`, an enum type, another annotation type, or a `Class`.

2. An element cannot declare any parameters.

3. An element cannot have a throws clause.

4. An element cannot define a type parameter.

The element `trust()` of the access control annotations is of type `Class` and hence, its existence is verified in compile time. However, from the previous rules [3], it is not possible to use the class `java.lang.reflect.Method` as the type of an annotation's element and hence the method is specified through its fully qualified name (`String`):

```
package mypackage;
public class A {
   @Invokers({ "mypackage.A.fooLegalInvoker()" })
```

```
    public void foo() {}
    public void fooLegalInvoker() {}
}
```

The previous code defines the method `fooLegalInvoker()` as the only allowed method to invoke `foo()`.

In the Zás framework, there are annotations for specifying the allowed invokers for the invocation of methods, invokers for modifying the value of a field, and also invokers for reading the value of a field.

In the current implementation, we assumed that it does not make sense to prevent the access with a strict set of invokers and, at the same time, specify the required abstract access modes to access that protected object: when a protected object declares the set of allowed invokers, only a method belonging to that set may be granted access. Hence, the invokers annotations were kept separated from the access control annotations and they cannot be used for a protected object at the same time:

```
// Compile time error: both annotations specified.
@Invokers({"mypackage.A.fooRightInvoker()"})
@AccessControlled("foo") public void foo() {}
```

This feature may not be directly related to access control because it does not make any use of the authenticated subjects, but because of its usefulness, and because it is also a way to control whose methods are allowed to access a protected object, we included it in the Zás framework.

Due to time constraints, the current version of Zás does not support the specification of the invokers in the Zás XML policy file.

4.6.2.6 Declaration of abstract access modes

The required abstract access modes for accessing the protected objects in Zás are `String`'s, which cannot be matched with the abstract access modes associated to the subject in a given client application. Hence, it is difficult to be actually sure one is using the appropriate abstract access modes in the definition of the access control requirements.

The Zás framework supports the declaration of the names of the abstract access modes, through the use of the annotation `@DeclareAbstractAccessMode`, whose default value is `''#''`, meaning that the protected object signatures are known by default.

```
@Retention(RetentionPolicy.RUNTIME) @Target(ElementType.TYPE)
@Inherited public @interface DeclareAbstractAccessMode {
   String[] value() default "#";
}
```

Therefore, Zás may check if the name of an abstract access mode has been declared and, if not, warn the programmer.

To be able to declare the abstract access modes in a given client application, we extend the abstract aspect `AbstractAccessModeDeclarationVerifier`. The abstract aspect must be parameterized with a class, created on the client code, which represents the default class where the abstract access modes are declared. Therefore, that class must be annotated with `@Declare-AbstractAccessMode`[10]:

```
public aspect AccessModeDeclarator extends
AccessModeDeclarator<DefaultDeclarator> {
   public Class<DefaultDeclarator> getDefaultDeclaratorClass() {
      return DefaultDeclarator.class;
   }
}

@DeclareAbstractAccessMode( { "foo", "bar" })
public class DefaultDeclarator {
}
```

In the previous snippet we declared the names of two abstract access modes: `foo` and `bar`. The `@DeclareAbstractAccessMode` may only be used in two places:

1. In the default declarator class, as specified previously, which is loaded at the beginning of the application as well as its abstract access modes, and

[10] This class cannot be extended because it would not be possible to load all subclasses and hence, it would not be possible to load the abstract access modes declarations of its subclasses.

2. In types containing protected objects, which are loaded at the moment an access attempt to a protected object is performed and hence, its abstract access modes declaration may be loaded:

```
package mypackage;

@DeclareAbstractAccessMode( "foo" )
public class A {
   @AccessControlled("foo")
   public void foo() {}
}
```

In the declaration of an abstract access mode, the package name cannot be included. Each declared name has a simple name, `foo` in the previous example, which may be used in the class and subclasses only, and a fully qualified name `mypackage.foo`. Hence, the collision of names are avoided.

Unlike the default abstract access mode declarator, which does not require any import abstract access modes statement to be used, all other abstract access modes declarations need to be explicitly imported, similarly to the import of classes and packages in Java:

```
package myotherpackage;

@ImportAbstractAccessModeDeclaration(mypackage.A.class)
public class B {
   // fully qualified name of the abstract access mode declared on A.
   @AccessControlled("mypackage.A.foo")
   public void foo() {}
   // simple name of the abstract access mode declared on A.
   @AccessControlled("foo")
   public void foo(String s) {}
}
```

Class `B` requires the import annotation (`@ImportAbstractAccessModeDeclaration` to make the abstract access modes declared in class `A` available in `B`. After that statement, the developer may use the simple name of abstract access mode or its fully qualified name, though we

recommend the latter, to prevent names collisions.

4.6.3 Future

New features, either partially implemented or not implemented at all, as well as important changes in the current ones, were already detected. A new version of Zás with several changes in the current features, as well as additional features, is already being implemented, and its code may be found in the project's web site at `http://zas.cvs.sourceforge.net/zas/`.[11] The case study presented in Chapter 5 was performed without any of the changes we present here:

1. Deal with, at least, two kinds of subjects: human subjects, represented as users, for example, and code subjects, represented as methods – The main idea is to allow methods to be subjects, having their own abstract access modes that grant them privileges to access other methods. E.g., if there was a subject $s1$ (an authenticated user, e.g.) that invokes `foo()` in the following example, who has the abstract access mode `foo` and not `bar`,

```
public class MyClass {
   @AccessControlled("foo")
   public void foo() {
      new MyClass2().bar();
   }
}
public class MyClass2 {
   @AccessControlled("bar")
   public void bar() {
   }
}
```

   We get an authorization exception as the result of the access attempt. However, a new annotation in the Zás framework, `@Privileges` exists to make methods be subjects as well:

```
public class MyClass {
```

[11] See the branch "B_zas-new-annotations".

```
    @AccessControlled("foo")
    @Privileges("bar")
    public void foo() {
       new MyClass2().bar();
    }
}
```

Hence, after the successful invocation of the method `foo()` by the subject $s1$, the current subject is the method `foo()` itself, which has the required abstract access mode to legitimately invoke `bar()`. After the invocation of `bar()`, the current subject is $s1$ again.

This is possible with the creation of a stack of subjects, Figure 4.5. Since it is important for us not to force developers to use a specific type of class to represent a non-method subject, our stack of subjects is constituted by method subjects only, which are instances of a final class `MethodSubject`, whose instances are created from the Zás framework, when a method declares itself as having privileges.

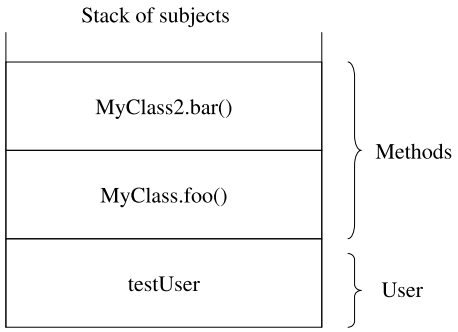

Figure 4.5: The concept of stack of subjects.

This feature, however, needs further study related to who may be responsible for the attribution of the abstract access modes to methods. Should a programmer be allowed to perform that task? Should it be the security manager, and the security manager only?

2. Depth verification – The depth feature in Zás, which controls the access control verification

depth is limited since the depth in effect at a specific point of the stack trace is based exclusively on the depth declared by the first method having access control in the current stack. This means that a programmer may define deep access control verification for an access controlled method `subFoo()` which, because it was invoked through an access controlled method `foo()` with shallow access control verification, does not have effect, since shallow will be in effect till the execution of `foo()` ends. A better solution would use a stack of invocations, associated to the current thread, which Zás would manage. To make the implementation easier and more readable, we created two new annotations, `@Shallow`, and `@Deep`, which replace the element `depth()` in `@AccessControlled`. At each moment, there would be a stack with the depth of access for each thread (by default, the stack has always an element of type `@Deep`). After the verification of a given method with access control, having one of the annotations `@Deep` or `@Shallow`, the access control verification depth is registered on the stack, specifying that all protected objects which are accessed from that method on, will use the depth defined in the top of the stack. The registration of the depth of the stack must happen after verifying the access, to ensure that the execution of the current method uses the depth defined in lower levels of the current stack. This implementation uses a new annotation `@Forced`, which replaces the element `suspicious()` from `@AccessControlled`, whose semantics is forcing the verification of the access for the current method, without changing the depth in the depth stack, however, meaning that only the current method should be checked but not any other protected object accessed within its flow of execution. To summarize, by default any access attempt is performed in a deep way. However, with the use of `@Shallow`, and `@Deep`, the depth of access control for the flow of execution of a given method may be changed. This feature has already been fully implemented in the new version of Zás.

3. Authorizations policy – XACML and Ponder are powerful languages to define authorizations, and hence decider classes capable of consulting the authorizations policy written in those languages should be provided with the Zás framework.

4. Certified source code – The JAAS model for certifying source code should be used as the way to "authenticate" source code. This would greatly enforce the application security, since external code accessing the application, which is outside the application's developers control, would also be subject to access control. This has not been explored at all.

Chapter 5

Evaluation and metrics

This work has focused on the use of AOP to modularize application-level security solutions. The use of this approach, as stated by De Win [47], requires a radical change in current development practices, since a secure application becomes a composition of loosely coupled components from a variety of stakeholders (business, security and so forth) that are put together in order to form an executable artefact.

Such a change must provide developers with huge benefits. In this section we try to address the benefits and drawbacks from the use of AOP for the implementation of security concerns. To accomplish this task, we have used Zás in a very large application, called Fénix. The results are presented next.

5.1 Fénix

Fénix is a Web application for managing academic institutions. It is composed of 6210 java classes (including almost 600 domain classes, persisted to a MySQL relational database containing 424 tables), 2348 JavaServer Pages (JSP) files, 299 XML files, and 185 properties files.

Fénix has been chosen for several reasons:

1. It is representative of a large class of applications, since it is a distributed application object-

oriented, Web application,

2. Its dimension is in itself a challenge to the application of Zás. If Zás can be applied with success to Fénix, it may safely be assumed that it can be applied to a wide range of applications.

The source code of the case study may be found in http://zas.cvs.sourceforge.net/zas/case-
In that location, there will be two folders, one containing the latest version of the case study and the other containing the first major application of Zás to Fénix, whose results we present in Section 5.2.5.

Fénix uses the Struts[1] and JavaServer Faces (JSF) frameworks. Domain modeling and decomposition are supported by a Domain Modeling Language (DML) [12] which developers use to define the static model of the domain in what may be seen as a limited textual form of UML. E.g.:

```
// ...
class net.sourceforge.fenixedu.domain.Employee extends
     net.sourceforge.fenixedu.domain.DomainObject {
   Boolean active;
   // ...
}
class net.sourceforge.fenixedu.domain.Person extends
     net.sourceforge.fenixedu.domain.Party {
   // ...
}
relation EmployeePerson {
   net.sourceforge.fenixedu.domain.Employee playsRole employee;
   net.sourceforge.fenixedu.domain.Person playsRole person;
}
// ...
```

DML parser reads the domain model file and generates the domain classes, including persistence code that makes use of the ObJect Relational Bridge (OJB) Object-Relational Mapping

[1] See http://struts.apache.org/ for details.

(ORM) tool from Apache.[2]

Fénix has a transaction system, which is managed by code generated from the DML model. Write transactions, which persist data, are allowed only in the flow of control of a given service. Hence, code outside of the flow of control of services may read data only, thus a service must be invoked in order to execute write operations from the data layer.

The abstract access modes from Fénix are textual representations of the system roles associated to each system user, i.e., the names of the roles. E.g., the system role `TIME_TABLE_MANAGER` that grants access to the management of the time tables was represented with the abstract access mode `TIME_TABLE_MANAGER`, which corresponds to the name of the role:

```
public class ChangeStudentsShift extends Service {
   @AccessControlled("TIME_TABLE_MANAGER")
   public void run(IUserView userView, Integer oldShiftId,
      Integer newShiftId, Set<Registration> registrations)
      throws ExcepcaoPersistencia, FenixServiceException {
      // ...
   }
}
```

Our main goal with the application of Zás to Fénix was to minimize the changes in the existing code. Hence, this application was a matter of replacing the existing access control related code, with the less changes possible, and with no changes in the functionalities. This means that the application of Zás was made *a posteriori* and not *a priori*, meaning that not all advantages of the framework were explored in this case study.

The use of Services Oriented Architecture (SOA) lead to the duplication of the functional code and, therefore, the Fénix project has evolved to a new more Domain Driven (DD) architecture. Hence, the access control should be applied to the domain classes, where the system's protected objects are. However, we did not want to add the currently logged user as a parameter to the domain classes, since we aimed at separating access control code from functional code.

Since Fénix already had its own authentication functionality, it was sufficient to develop a

[2] See `http://db.apache.org/ojb/` for details.

mechanism to obtain the authenticated user associated with each session. This is simply a matter of defining the methods `currentSubject()` and `defaultDeciderClass()` in the concrete aspect `FénixAccessController`, thus implementing the corresponding abstract methods from the abstract aspect `Enforcer`,

```
public aspect FenixEnforcer extends
   Enforcer<IUserView> {

   public IUserView currentSubject() {
      return AccessControl.getUserView();
   }
   public Class<FenixDecider> defaultDeciderClass() {
      return FenixDecider.class;
   }
   // In Fénix we protect the executions, and not the calls
   public pointcut accessToControlledMethods() :
      execution(@AccessControlled !private *
         net.sourceforge.fenixedu..*(..));

   // In Fénix we protect the executions, and not the calls
   public pointcut accessToControlledConstructors() :
      execution(@AccessControlled !private
         net.sourceforge.fenixedu..new(..));
}
```

The method `currentSubject()` was implemented as a simple invocation to the method `getUserView()` from the Fénix application which returns the instance of the subject associated to the current HyperText Transfer Protocol (HTTP) session. The `defaultDeciderClass()` simply returned the class that implements the default decision function in Fénix:

```
public class FenixDecider extends Decider<IUserView> {
   public Collection<String> currentSubjectAccessModes() {
      final Set<String> abstractAccessModes = new HashSet<String>();
      final IUserView userView = getCurrentSubject();
      if (userView != null) {
         for (final RoleType roleType : userView.getRoleTypes()) {
            abstractAccessModes.add(roleType.getName());
         }
```

```
        }
        return abstractAccessModes;
    }
}
```

The default decider must implement the abstract method `currentSubjectAccessModes()` from the abstract class `Decider`, which iterates over the system roles associated to the current subject, and adds its name, which is unique in the system, to the abstract access modes associated to that subject.

About 70% of the access control related code existing in Fénix was changed to use Zás. In most cases the change simply meant removing the access control filters from the services,

```
<!-- Before Zás -->
<service>
    <name>ChangeStudentsShift</name>
    <implementationClass>
        net.sourceforge.fenixedu.applicationTier.Servico.
            sop.ChangeStudentsShift
    </implementationClass>
    <description />
    <isTransactional>true</isTransactional>
    <filterChains>
        <chain name="TimeTableManagerAuthorization" />
    </filterChains>
</service>

<!-- With Zás -->
<service>
    <name>ChangeStudentsShift</name>
    <implementationClass>
        net.sourceforge.fenixedu.applicationTier.Servico.
            sop.ChangeStudentsShift
    </implementationClass>
    <description />
    <isTransactional>true</isTransactional>
</service>
```

and the addition of the necessary access control annotations, either through ITD or directly to

the protected object:

```
@AccessControlled("TIME_TABLE_MANAGER")
public class ChangeStudentsShift extends Service {
   public void run(IUserView userView, Integer oldShiftId,
      Integer newShiftId, Set<Registration> registrations)
      throws ExcepcaoPersistencia, FenixServiceException {
      // ...
   }
}
```

For domain level access control, Fénix uses a bytecode manipulation mechanism in what amounts to ad hoc weaving. Thus, Fénix implements a limited type of AOP in Java. Zás implements essentially the same idea, though using AspectJ constructs. The advantage of using AspectJ in terms of robustness, maintainability, expression power, etc., are considerable, as will be discussed. Additionally, Zás is a reusable framework providing features such as shallow or deep access control, and privileged access to protected objects which the bytecode manipulation solution, developed in an ad hoc way, did not possess, of that would require considerable effort to implement.

The conversion from the original bytecode manipulation solution to Zás involved replacing the annotation `@Checked` with annotation `@AccessControlled`,

```
// before Zás
@Checked("SpacePredicates.
   checkIfLoggedPersonHasPermissionsToManageResponsabilityUnits")
public void setSpaceResponsibilityInterval(YearMonthDay begin,
   YearMonthDay end) {
   checkSpaceResponsabilityIntersection(begin, end,
      getUnit(), getSpace());
   super.setBegin(begin);
   super.setEnd(end);
}
// with Zás
// @Checked("SpacePredicates.
// checkIfLoggedPersonHasPermissionsToManageResponsabilityUnits")
@AccessControlled(deciderClass = SpacePredicates.
   CheckIfLoggedPersonHasPermissionsToManageResponsabilityUnits.class)
```

```
// ... as before
```

and the creation of wrapper classes around the existing decider classes, applying the *strategy* design pattern [22],

```
public abstract class
   PredicateBaseDecider<T extends DomainObject>
   extends FenixDecider {

   protected abstract AccessControlPredicate<T> getPredicate();
   @Override
   public boolean hasAccess(String protectedObjectAccessModes,
      JoinPoint joinPoint, StaticPart enclosingStaticPart) {
      return getPredicate().evaluate((T) joinPoint.getTarget());
   }
}
```

The abstract class `PredicateBaseDecider` was created to simplify the integration with the existing decider classes. The class provides an abstract method `getPredicate()` which returns an instance of the Fénix class `AccessControlPredicate` responsible for evaluating the legitimacy of the accesses. That class must be extended and its abstract method implemented appropriately, as follows,

```
public class SpacePredicates {
   // ...
   // previously provided by Fénix.
   public static final AccessControlPredicate<SpaceResponsibility>
      checkIfLoggedPersonHasPermissionsToManageResponsabilityUnits =
      new AccessControlPredicate<SpaceResponsibility>() {
      public boolean evaluate(SpaceResponsibility spaceResponsibility) {
         spaceResponsibility.getSpace().
            checkIfLoggedPersonHasPermissionsToManageSpace(
            AccessControl.getPerson());
         return true;
      }
   };
   // addition because of Zás.
   public class
      CheckIfLoggedPersonHasPermissionsToManageResponsabilityUnits
```

```
        extends PredicateBaseDecider<SpaceResponsibility> {
        @Override protected
           AccessControlPredicate<SpaceResponsibility> getPredicate() {
           return checkIfLoggedPersonHasPermissionsToManageResponsabilityUnits;
        }
     }
     // ...
}
```

During the changes related to the access control we performed in Fénix, the following limitations were detected in the Zás framework:

1. controlling the access to the object returned by a method, and not the ability to invoke it. This implies the unconditional invocation of the method and the checking of the returned method, naturally *after* its execution. This problem may be solved in a future version of the Zás framework with the creation of a new annotation, e.g., `ReturnedObjectAccessControlled`, and the use of *around* advices,

2. attach different access control requirements to the same method, according to the different logic names which may be used to invoke it. In Fénix, the invocation of the `run()` method of a service is accomplished indirectly through the specification of the name of the service:

   ```
   ServiceUtils.executeService(userView, "ChangeStudentsShift", new
   Object[] { userView, oldShiftId, newShiftIf, registrations });
   ```

   The invocable services are specified in a configuration file:

   ```
   <service>
      <name>ChangeStudentsShift</name>
      <implementationClass>
         net.sourceforge.fenixedu.applicationTier.Servico.
            sop.ChangeStudentsShift
      </implementationClass>
      <!-- ... -->
   </service>
   ```

   This limitation could be fixed through the creation of a mechanism that associates a logical

name with a set of access control rules. A possibility is presented in the next snippet of code, which defines that the invocation of the service using the name `ChangeStudentsShift` requires the subject to have the abstract access mode `foo` while the name `ChangeStudentsShift2` requires `bar`,

```
@AccessControlledService(
   names = { "ChangeStudentsShift", "ChangeStudentsShift2" },
   requirements = { @AccessControlled("foo"), @AccessControlled("bar") }
)
public class ChangeStudentsShift extends Service {
   public void run(IUserView userView, Integer oldShiftId,
      Integer newShiftId, Set<Registration> registrations)
      throws ExcepcaoPersistencia, FenixServiceException {
      // ...
   }
}
```

5.2 Analysis

As stated by Baldwin in [5], an AO approach is of great use because of its advantages in terms of maintenance of code. To measure its benefit, e.g., we may use the Net Option Value (NOV) formula. While in OO approaches, one needs N changes in order to implement or change a crosscutting concern,

$$\text{NOV} = \text{Change benefit} - N \times \text{Change costs},$$

where N is the number of locations in the code where that concern is reflected. In AOP, generally, it is necessary only one change,

$$\text{NOV} = \text{Change benefit} - 1 \times \text{Change costs},$$

Also according to Baldwin [5], another huge benefit of AOP is reusability since the improved modularity aspects provide, make it possible for the aspect code to be reused in several different applications.

That being said, we will present the results of the application of AOP and specifically the Zás framework to Fénix, using the following metrics:

1. Number of files
2. Lines of Code (LOC)
3. Performance
4. Percentage of affected access controlled methods
5. Access Control Requirements Specification Ratio (ACRSR)

The accuracy of this analysis is limited, though, since it did not fully replace the existing access control mechanisms due to time constraints. Additionally, and since that the Zás framework is for developers, this measurement accuracy is also limited because there are no other significant experiences of the application of Zás in other applications, by developers not being the creator of Zás. Finally, the impact of Zás in Fénix, as shown next, is not very significative because one of the existing access control mechanisms was already based on weaving, though an ad-hoc weaving mechanism, and the services access control mechanism enables the separation of concerns.

5.2.1 Number of files

The total number of files in the project before and after the application of Zás, see Table 5.1, is one of the possible ways to evaluate the quantity of changes that one needs to do in order to apply Zás.

The application of the framework allowed to reduce the number of files, still keeping the same functionalities and access control level. Decreasing the total number of files is important since the smaller the application, the smaller will be the maintenance effort. In this case, however, the reduction in the total number of files was not meaningful.

	Fénix		
File	Before	Zás	Diff
Java	6210	6205	−5
Aspect	0	1	+1
XML	299	299	0
Properties	185	185	0
Total	6133	6108	−4

Table 5.1: Number of files before and after Zás.

5.2.2 LOC

As we can see in Table 5.2, the solution based on Zás allowed to decrease the total number of lines of code, making the application easier to maintain and manage.

	Fénix		
File	Before	Zás	Diff
Java	756061	757287	+1226
Aspect	0	39	+39
XML	78401	76103	−2298
Properties	32425	32425	0
Total	866887	865854	−1033

Table 5.2: Lines of code before and after Zás.

Nevertheless, the application increased in terms of Java code since most access control in the access control filters associated to the Fénix services were removed from the XML and added in the Java code.

Since Zás did not fully replace the existing access control filters and there are dependencies between them (an access control filter in Fénix is often composed by several other filters), many

lines of code and the corresponding Java files were not removed.[3]

The use of the AOP features such as ITD makes the capabilities of the IDE the developers use very important. E.g., the AspectJ Development Tools (AJDT) plugin for Eclipse, which shows when a given piece of code is advised by an aspect is of extreme use and almost a requirement for AOP programmers. However, we faced several problems with the memory required by the AspectJ compiler to build the application in Eclipse. Therefore, we added the access control requirements for each protected object directly with the protected object, instead of using ITD to inject the access control requirements, and hence be able to look at the source code and know exactly what is the access control requirements for each protected object.

5.2.3 Performance

To measure the impact of using Zás in the application performance, the execution and compile times were compared.

The compile time was obtained by calculating the average of the elapsed time of 30 independent compilations. The execution time was obtained by calculating the average of the elapsed time of 30 executions of a small test application which invokes repeatedly methods which are subject of access control either in the domain level and on the services level. The number of invocations varied from 150 to 2250, with a step of 150, for the domain level, and from 50 to 750, with a 50 step, for the services level.

These tests tried to measure the gap between both solutions, and the repetition of each experiment with an increasing number of invocations tried to verify if a potential gap gets better throughout the time.

The tested methods were chosen randomly, among all methods that started using Zás as the access control mechanism. These methods were chosen because they did not take much time while being executed to not affect the performance analysis of the access control mechanisms.

[3] We expect to greatly decrease the number of lines and files in Fénix, though, when the existing access control mechanisms are fully replaced by the Zás framework.

One better approach for choosing the tested methods would include a previous study of the most used services and functionalities from the system and test those specific services according to its use and thus provide a more concrete vision of the global impact in the system as a whole. This previous study, however, would be hard to accomplish in a short period of time and hence was not performed.

The tests were performed in an Intel Pentium IV Hyper-Threading (HT) with 3.06 GHz and 2 GiB of Random Access Memory (RAM) memory.

5.2.3.1 Protected domain

The tested method simply sets the code of a given course. The solution before Zás presents a better performance, especially with a small number of invocations, as may be seen in Figure 5.1.

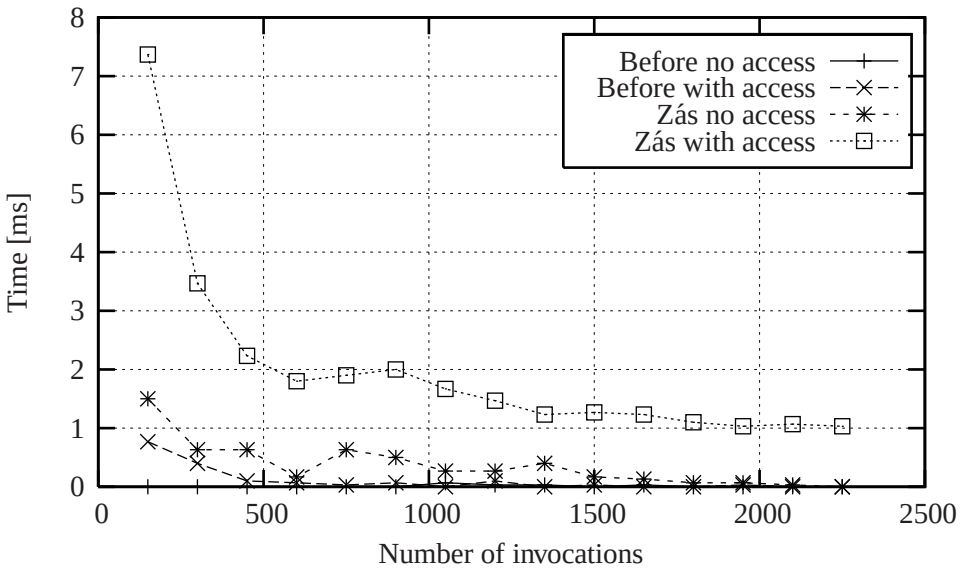

Figure 5.1: Fénix domain level performance before and after Zás.

The solution based on Zás is affected with its initialization, thus making a significant different after the first invocations to the domain method which, however, tends to decrease throughout the time.

5.2.3.2 Protected services

The service we chose simply returns the homepage of the currently logged in user, if there is one or creates a new homepage for that user. Figure 5.2 summarizes the results.

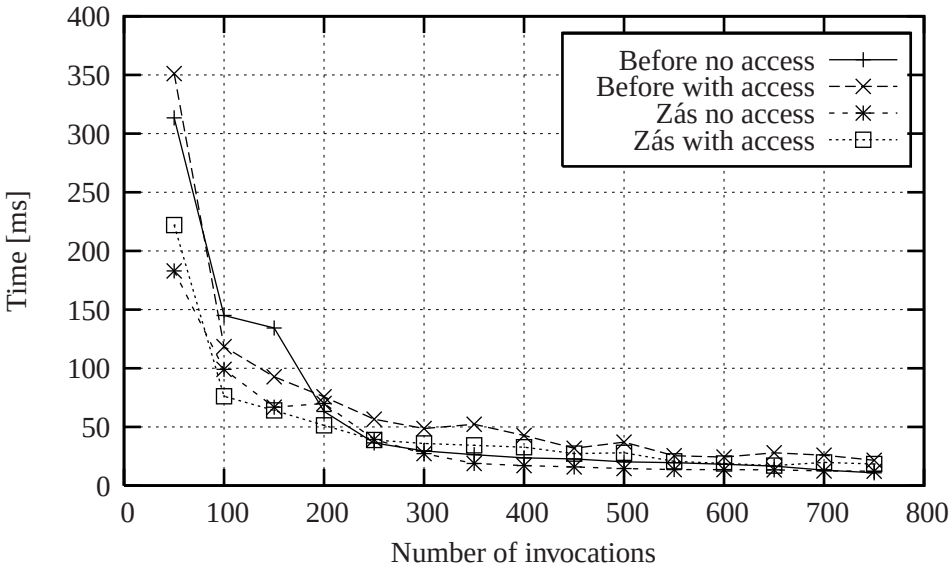

Figure 5.2: Fénix services level performance before and after Zás.

The solution based on Zás is faster, especially when comparing accesses for users not having access to the protected object. Throughout the time, however, the differences tend to become smaller.

5.2.3.3 Compilation performance

The compile time has a direct impact on the development. Figure 5.3 presents the results.

Compiling a very big application such as Fénix with aspects is a problem in what concerns the compilation time. With aspects, the application took about 71 seconds longer due to the weaving process, which corresponds to an increase of about 30%.

Since the Zás framework modularizes only non functional requirements, the normal development process in Fénix could use the compilation of the functional code, using the Java compiler,

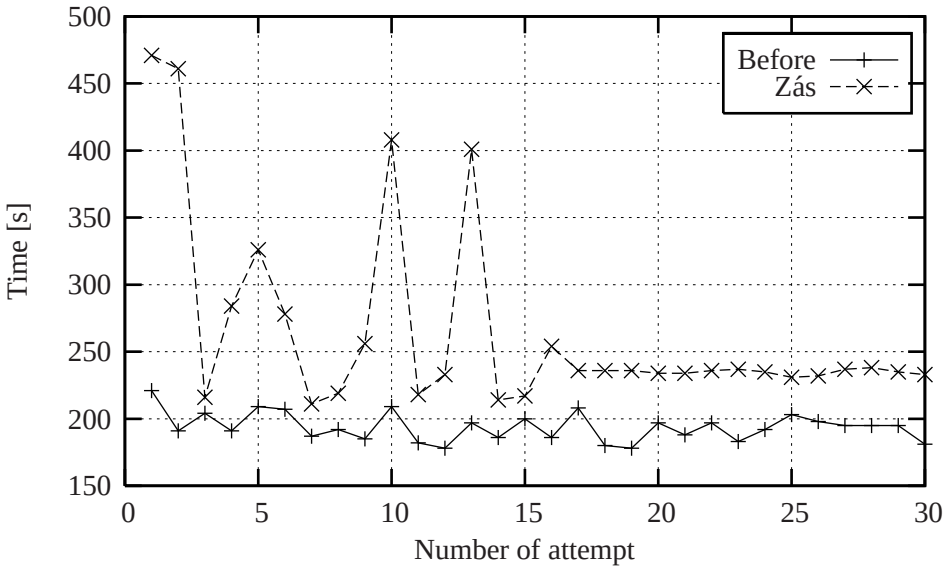

Figure 5.3: Fénix compilation time before and after Zás.

therefore having smaller cycles of coding, compilation and tests. The compilation with Aspects would be relegated to the regression tests and to the occasions where the access control itself would be in development or maintenance.

5.2.4 Percentage of affected access controlled methods

As stated by Lopes in [30], "one way of measuring the code tangling is by counting the number of methods affected by aspect code". The number of affected methods in the total shows how much access control-related code was necessary for each application and thus represent how difficult was to add access control-related concerns for each application. For the total of methods we have counted the number of methods in those classes that were supposed to have access control. Table 5.3 summarizes the results.

The access control functionalities were implemented in about 70% of the situations (1422 methods requiring authorization versus 1021 methods for security purposes implemented with Zás). With the new development philosophy within Fénix, the domain will become richer and

	Fénix	
Layer	To control	With Zás
Domain	153	153
Service	1269	868
Total	1422	1021

Table 5.3: Percentage of affected access controlled methods.

security concerns will be added there, closely to the data. We have implemented a considerable part of the access controlled methods to conclude that Zás addresses the majority of the situations requiring access control enforcing.

5.2.5 ACRSR

The ACRSR is a metric we propose to measure the effort for the specification of the access control requirements for each application. This ratio relates the number of access control requirements rules specified with the number of affected points. This metric is a percentage in which, the higher its value, the greater the effort of the access control requirements specification. The formula for this metric is as follows:

$$\text{ACRSR} = 100 \times \frac{\text{Number of Rules}}{\text{Affected Methods}}$$

Due to the limitations in terms of using the AJDT plugin capabilities to show the developer the methods that are being advised and which ones are being annotated, we decided to mark directly the protected objects with the access control annotations. Nevertheless, in an early application of Zás to Fénix, in February, we both directly annotated the protected objects and used ITD to specify the protected objects and its access control requirements. Table 5.4 summarizes the results.

This early application of Zás to Fénix lead us to some interesting conclusions. E.g., there was a bug with the order of annotations injection, which has been submitted to the AspectJ compiler

	Fénix
Number of rules	637
Affected methods	921
ACRSR	69.2%

Table 5.4: Access Control Requirements Specification Ratio.

community[4]. This bug forced us to directly mark the non access controlled "resources", since it is not possible to control the order of the annotations injection. We expect the ACRSR to improve when that bug is fixed and we are able to control the order of the annotations injection. Additionally, there were services inside package such as `coordinator` in which only a user having the abstract access mode `MANAGER` could execute. Those classes could be refactored to the package `manager`, thus greatly reducing the effort for the specification of access control rules.

5.2.6 Conclusions

From the evaluation analysis we performed in this section, we believe that Zás has potential to be the dominant access control mechanism in Fénix and to fully replace the existing mechanisms, since it permits to use a single access control model, in both the services and domain layers of the application, which is something we desire in Fénix.

The main problems we detected with the use of AOP in Fénix were related to the performance of the system: executing code, in which the Zás-based version presented better performance in the services layer, but a worse performance in the domain layer, and the compilation time which, however, when used to implement non-functional requirements, as what happens with Zás, may be solved with different compilation processes: the normal development process in Fénix could use the compilation of the functional code, using the Java compiler, therefore having smaller cycles of coding, compilation and tests. The compilation with Aspects would be relegated to the regression tests and to the occasions where the access control itself would be in development or

[4]Bug 169699, https://bugs.eclipse.org/bugs/show_bug.cgi?id=169699.

maintenance.

The specification of the protected objects and their access control requirements through aspects that inject those requirements have the advantage of the non-invasiveness. However, for very large projects, in which the use of the IDE features is difficult or even impossible, adding the access control annotations close to the protected objects is a better choice, since it adds more expressiveness to the code. Additionally, the compilation time tends to increase with the injection of annotations, especially when using wildcards in the definition of the pointcuts, and it also gets harder to control the scope of the affected resources.

Since Zás is a framework, it hides the implementation details from the developers who use it. Therefore, earning new capabilities and knowledge related to AOP by the technical team is not a requirement, since the team just need to use Java constructs – annotations – to specify the protected objects' access control requirements. We based these ideas in our experience from the application of Zás to Fénix, which is in production at ISCTE[5], since February of 2007.

Nevertheless, the application of Zás to Fénix requires more study and analysis, namely in what concerns the use of the Zás framework advanced features. That will be relegated to further work.

[5]Instituto Superior de Ciências do Trabalho e da Empresa – http://iscte.pt/.

Chapter 6

Conclusions and further work

In this work a new AO access control framework based on the abstract reference monitor [2] was proposed. To add authorization concepts in existing applications, the Zás framework explores the capability to specify meta-information through the usage of Java 5 annotations and XML. It does so in an easy, expressive and non-invasive way. Even though the framework is at its early stage of development, it has already shown the potential for the use of AOP to modularize access control, making it simple to implement, support and configure, and even more flexible. Zás is also dynamic, allowing changes in runtime to the access control requirements associated to the protected objects. The framework usage, whose development was motivated by a proposal by Laddad [26], reduces the scattering of the code all over the application, as well as its entanglement with the functional code.

De Win *et al.* [48] criticize AspectJ to implement security concerns, ironically due to its excessive flexibility, because security is a very rigid and strict concept. Under this point of view, Zás is very similar to AspectJ, because it is also very, perhaps too much, flexible: its misuse may lead to security leaks. Due to this reason, it is necessary a deeper study and analysis, namely to the best practices and potentially new requirements to implement in order to control the flexibility of Zás in client applications.

In the near future, we intend to improve the framework, supplying more technical documen-

tation and a manual of best practices, bearing in mind the insight gained with its application to several applications. Nevertheless, some points requiring further investigation have already been identified. How does Zás behaves when other crosscutting concerns are implemented using aspects? How do we deal with potential conflicts between different aspects, particularly if those have the potential of affecting the access control? Nakajima and Tamai [32] have proposed a technical analysis to verify the coherence between the authorizations policy and the application code. The proposal, however, assumes that the policies are static. How could that technique be applied in the case of dynamic policies, as allowed by Zás? Currently, authorizations are specified in Java code. It is important that, in the future, the authorizations are specified using a domain specific language, such as XACML.

The framework deals with the inheritance of access control requirements defined in classes and interfaces to their fields and methods. However, it does not deal with questions related to the extension of classes and with the implementation of interfaces, i.e., with the polymorphism inherent to OO applications. Should a subclass redefine freely the access control requirements of a method that overrides the ones from the base class? Should it, if necessary, release that method of any access control? Should it use, by default, the access control requirements defined on the super method, or the ones defined in the current method, i.e., should the access control requirements be defined as a disjunction between the access control requirements defined on the super method and the ones defined in the current method? Should the access control requirements associated to methods be treated as their pre-conditions?

Currently, Zás does not address any of the previous questions. It may support all or at least some of them in the future, helping the programmer to decide what to do in each situation, through the introduction of new types of meta-information about the access control requirements, such as `@OverrideAccessControl` to state that a method `foo()`, overridden in a given subclass, fully replaces any access control requirements defined on its super method, `@InheritsAccessControl` to inherit the access control requirements from the super method, causing a disjunction between the requirements defined on super and on the current method, and `@RefineAccess-`

`Control` to cause a conjunction between the access control requirements defined on both methods, leading to any attempt to the method on the extending class to require the access control requirements both from the super method and from the method itself.

Bibliography

[1] Massimo Ancona, Walter Cazzola, and Eduardo B. Fernandez. Reflective authorization systems: Possibilities, benefits and drawbacks. In J. Vitek and C. D. Jensen, editors, *Secure Internet Programming*, volume 1603 of *Lecture Notes in Computer Science*, pages 35–50. Springer-Verlag, 1999. 1, 9

[2] James P. Anderson. Computer security technology planning study. Technical Report Volume II, Electronic Systems Division, Air Force Systems Command, Hanscom Field, Bedford, October 1972. 1, 7, 81

[3] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, Upper Saddle River, New Jersey, USA, 4th edition, August 2005. 55

[4] AspectJ Team. The AspectJ project at Eclipse.org. Web page. [2006-04-16]. 2, 45, 47

[5] Carliss Baldwin. The power of modularity: The financial consequences of computer and code architecture. In *Proceedings of the 5th international conference on Aspect-oriented software development*, pages 1–1, Bonn, Germany, March 2006. 71

[6] Abhijit Belapurkar. Java authorization internals – a guided tour of the Java 2 platform and JAAS authorization architectures. Web publication, May 2004. 10

[7] Elisa Bertino. Data hiding and security in an object-oriented database system. In *8th IEEE International Conference on Data Engineering*, pages 338–347, Phoenix, Arizona, USA, 1992. 40, 54

[8] K. Beznosov. *Engineering Access Control for Distributed Enterprise Applications*. PhD thesis, Engineering Access Control for Distributed Enterprise Applications, Miami, Florida, 2000. 8

[9] Ron Bodkin. Enterprise security aspects. In *Proceedings of the AOSD Technology for Application-level Security Workshop*, Lancaster, UK, 2004. 9

[10] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 2001. 26

[11] G. Bostrom. Database encryption as an aspect. In *Proceedings of the AOSD Technology for Application-level Security Workshop (AOSDSEC'04)*, Lancaster, UK, 2004. 9

[12] João Cachopo and Rito Silva. Combining software transactional memory with a domain modeling language to simplify web application development. In ACM International Conference Proceeding Series, editor, *6th International Conference on Web Engineering*, pages 297 – 304, New York, NY, UYSA, 2006. 64

[13] Soma Chaudhuri. Colloquium series spring 2005, 2005. [2006-11-09]. 1

[14] Michael Coté. JAAS book: Java authentication and authorization. Originally written for publication by Manning, [2006-04-16]. 2

[15] Carlos Nuno da Cruz Ribeiro. *Uma Plataforma para Poléticas de Autorização para Organizações*. PhD thesis, Universidade Técnica de Lisboa - Instituto Superior Técnico, 2002. 5, 8

[16] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *2nd International Workshop on Policies for Distributed Systems and Networks*, Bristol, U.K., January 2001. 5, 40

[17] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall, Inc., 1st edition edition, October 1976. 1

[18] Dulce Domingos. *Controlo de Acesso em Fluxos de Trabalho Adaptáveis*. PhD thesis, Department of Informatics, University of Lisbon, December 2005. DI/FCUL TR-05-26. 1, 24

[19] Eduardo B. Fernandez. Patterns for operating systems access control. In *9th Conference on Pattern Languages of Programs (PlLoP)*, 2002. 2

[20] Eduardo B. Fernandez and John C. Sinibaldi. More patterns for operating systems access control. In *Euro Pattern Languages of Programs (PLoP)*, 2003. 2

[21] David F. Ferraiolo, D. Richard Kuhn, Ramaswamy Chandramouli, and John Barkley. Role Based Access Control (RBAC). Web page, March 2006. [2006-03-08]. 6

[22] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994. 9, 69

[23] S. Gao, Y. Deng, H. Yu, X. He, K. Beznosov, and K. Cooper. Applying aspect-orientation in designing security systems. In *Proceedings of 16th International Conference on Software Engineering and Knowledge Engineering (SEKE'04)*, Banff, Alberta, Canada, 2004. 9

[24] Oddleif Halvorsen and Øystein Haugen. Proposed notation for exception handling in UML 2 sequence diagrams. In *Proceedings of the 2006 Australian Software Engineering Conference (ASWEC'06)*, 2006. 26

[25] Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley, 2004. 26

[26] Ramnivas Laddad. *AspectJ in Action*. Manning, Greenwich, Connecticut, USA, 2003. 2, 3, 11, 18, 40, 81

[27] Ramnivas Laddad. AOP@Work AOP and metadata: A perfect match, part 1). Web publication, March 2005. 41

[28] Ramnivas Laddad. Personal communication, 2006. 33, 45

[29] Charlie Lai, Li Gong, Larry Koved, Anthony Nadalin, and Roland Schemers. User authentication and authorization in the Java™platfom. In *Proceedings of the 15th Annual Computer Security Applications Conference*, Phoenix, Arizona, USA, December 1999. 2, 8

[30] Cristina Videira Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997. 77

[31] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, New Jersey, USA, 2nd edition, March 2000. 36

[32] Shin Nakajima and Tetsuo Tamai. Formal specification and analysis of JAAS framework. In *Proceedings of the 2006 International Workshop on Software Engineering for Secure Systems*, pages 59–64, Shanghai, China, May 2006. ACM, ACM Press. 82

[33] Dana Nourie and Mike McCloskey. Regular expressions and the Java programming language, August 2001. Updated April 2002. 44

[34] Scott Oaks. *Java Security*. O'Reilly, 2nd edition, 2005. 2, 17

[35] OASIS. eXtensible Access Control Markup Language XACML version 2.0. Technical report, OASIS, 2005. 5, 7, 40

[36] B. Broom P. Ashley, M. Vandenwauver. A uniform approach to securing unix applications using SESAME. In *Proceedings of the 3rd ACISP Conference*, pages 24–35, 1998. 8

[37] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998. 26

[38] Tatyana Ryutov and Clifford Neuman. The specification and enforcement of advanced security policies. In *Proceedings of the Conference on Policies for Distributed Systems and Networks (POLICY 2002)*, Monterey, California, 2002. 8

[39] V. Shah and F. Hill. An aspect-oriented security framework. In *Proceedings of DARPA Information Survivability Conference and Exposition (DISCEX'03)*, volume 2, pages 143–145, Washington, DC, USA, 2003. 9

[40] Chris Strahorn. Security in next-generation databases. In *Proceedings of Citeseer Digital Libraries*, nov 1998. 11, 12

[41] Sun Microsystems, Inc. Java 2 platform SE 5.0 API: Pattern class. [2006-04-16]. 44

[42] Sun Microsystems, Inc. Java technology: Security and the Java platform. [2006-04-16]. 15

[43] Roshan K. Thomas and Ravi S. Sandhu. Discretionary access control in object-oriented databases: Issues and research directions. In *Proceedings of the 16th NIST-NCSC National Computer Security Conference*, pages 63–74, Baltimore, MD, September 1993. 54

[44] Tine Verhanneman, Frank Piessens, Bart De Win, Eddy Truyen, and Wouter Joosen. A modular access control service for supporting application-specific policies. *IEEE Distributed Systems Online*, 7(6), June 2006. 2, 9

[45] J. Viega, J. T. Bloch, and P. Chandra. Applying aspect-oriented programming to security. *Cutter IT Journal*, 2001. 9

[46] Wikipedia, the Free Encyclopedia. Access control. [2006-10-29]. 1

[47] Bart De Win. *Engineering application-level security through aspect-oriented software development*. PhD thesis, Department of Computer Science, Catholic University of Leuven, Leuven, Belgium, 2004. 2, 9, 63

[48] Bart De Win, Frank Piessens, and Wouter Joosen. How secure is AOP and what can we do about it? In *Proceedings of the 2006 international workshop on Software engineering for secure systems*, pages 27–34, New York, NY, USA, 2006. ACM Press. 81

[49] Joseph Yoder and Jason Barcalow. Architectural patterns for enabling application security. In *PLoP'97, Proceedings of the 4th Conference on Patterns Language of Programming*, 1997. 2