Chapter 1

Introduction

This book is about aspect-oriented software development (AOSD), a set of emerging technologies that seeks new modularizations of software systems. AOSD allows multiple concerns to be separately expressed but nevertheless be automatically unified into working systems. We intend this book to be an overview of these technologies for the computer professional interested in learning about state-of-the-art developments.

In general, programming is about realizing a set of requirements in an operational software system. One has a (perhaps evolving) set of properties desired of a system, and proceeds to develop that system to achieve those properties. Software engineering is the accumulated set of processes, methodologies, and tools to ease that evolutionary process, including techniques for figuring out what we want to build and mechanisms for yielding a higher-quality resulting system.

A recurrent theme of software engineering (and engineering in general) is that of modularization: "separation and localization of concerns." That is, we have "concerns"—things we care about—in engineering any system. These concerns range from high-level, user-visible requirements like reliability and security to low-level implementation issues like caching and synchronization. Ideally, separating concerns in engineering simplifies system development by allowing the development of specialized expertise and by producing an overall more comprehensible arrangement of elements.

Traditional software development has focused on decomposing systems into units of primary functionality, recognizing the other issues of concern, and leaving it to programmers both to code modules corresponding to the primary functionality and to make sure that all other issues of concern are addressed in the code wherever appropriate. Sometimes these other concerns can be packaged into modules of behavior themselves (e.g., subroutines, methods, or procedures). However, often the degree of shared context or the cost of contextual change (for example, the cost of a subprogram call) necessitates intermixing (crosscutting) the instructions for the primary functionality and the other concerns. In any case, conventional development requires programmers to keep in mind all the things that need to be done, how to deal with each issue, the problems associated with the possible interactions, and the execution of the right behavior at the right time.

Spreading out the responsibility for invoking the code for multiple concerns to all programmers produces a more brittle system. Each programmer who has to do something right is one more person who can make a mistake; each spot where something needs to be done is a potential maintenance mishap. The distribution of the code for realizing a concern becomes especially critical as the requirements for that concern evolve—a system maintainer must find and correctly update a variety of (likely poorly identified) situations.

We use the term aspect-oriented programming (AOP) to describe the activity of programming with multiple crosscutting concerns or aspects. The general *modus operandi* of programming AOSD systems is to let system developers express the behavior for each concern in its own module. Such a system must also include some directions for how the different concerns are to be knitted together into a working system (for example, to which program entities each separate concern applies) and a mechanism for actually producing a working system from these elements. For example, many AOSD systems provide a way to say, "High security is achieved by doing *X*. Reliability is achieved by doing *Y*. I want high security in the following places in the code and reliability on these operations." The AOSD system then produces an object that invokes the high security and reliability codes appropriately.

Concern-level illustrations of the application of AOP techniques include replication, configuration, debugging, mobility, program instrumentation, security, code movement, and synchronization. AOP is only beginning to penetrate commercial applications, but interesting prototypes have been demonstrated in areas such as application servers, operating systems kernels, real-time distributed event channels, distributed middleware, distributed quality-of-service, multi-agent system architectures, object databases, domain-specific visual modeling, collaboration, workflow, e-commerce, software visualization, engineering design, and data processing.

1.1 BOOK ORGANIZATION

Just like object-oriented programming, aspect technology started with aspect-oriented programming languages. Currently, several aspect-oriented languages are in widespread availability, and researchers are continually inventing new ones. Part 1 of this book, "Languages and Foundations," examines this area, including descriptions of not only proposed AOP languages but also programming models based on aspect ideas and chapters discussing the fundamental and historical nature of AOP.

Just as object-oriented programming led to the development of a large class of object-oriented development methodologies, AOP has encouraged a nascent set of software engineering technologies. Part 2, "Software Engineering," examines these issues, include methodologies for dealing with aspects, modeling techniques (often based on the ideas of the Unified Modeling Language, UML), and testing technology for assessing the effectiveness of aspect approaches.

Of course, the ultimate aim of programming is to develop software systems. In Part 3, "Applications," we present descriptions of the application of aspect technology to particular software problems, including examples that range from the systems to application levels.

Each part of the book includes an introduction to that area and chapters by contributors describing their own work. We invited each contributor to either create an original chapter, targeted at the advanced programmer, or to nominate reprinting an existing paper meeting those criteria. Several of our contributors chose to meld the two approaches, revising existing work in light of new experience and the intended audience.

AOSD is a rapidly evolving area. This format has enabled us to present the reader with a wider overview, a more current set of work, and a clearer sense of the diversity of opinions than a synopsis of different research or an in-depth study of one particular research direction would have provided.

1.2 COMMON TERMINOLOGY

Certain common terminology and themes pervade these papers. Different authors have variations on the meaning they assign to these ideas. This is a symptom of intellectual youth and ferment. They use certain terms freely. Thus, it is helpful to begin with a brief glossary of common AOSD concepts.

Concerns. Any engineering process has many things about which it cares. These range from high-level requirements ("The system shall be manageable") to low-level implementation issues ("Remote values shall be cached"). Some concerns are localized to a particular place in the emerging system ("When the M key is pressed, the following menu shall pop up"), some refer to measurable properties of the system as a whole ("Response time shall be less than a second"), others are aesthetic ("Programmers shall use meaningful variable names"), and others involve systematic behavior ("All database changes shall be logged"). Generically, we call all these concerns, though AOSD technology is particularly directed at the last, systematic class.

Crosscutting concerns. Software development addresses concerns, both concerns at the user/requirements levels and at the design/implementation level. Often, the implementation of one concern must be scattered throughout the rest of an implementation. We say that such a concern is *crosscutting*. Note that what is crosscutting is a function of both the particular decomposition of a system and the underlying support environment. A particular concern might crosscut in one view of an architecture while being localized in another; a particular environment might invisibly support a concern (for example, security) that needs to be explicitly addressed in another.

Code tangling. In conventional environments, implementing crosscutting concerns usually results in code tangling—the code for concerns becomes intermixed. Ideally, software engineering principles instruct us to modularize our system software in such a way that (1) each module is cohesive in terms of the concerns it implements and (2) interfaces between modules are simple. Software that complies with these principles tends to be easier to produce, more naturally distributed among different programmers, easier to verify and test, and easier to maintain, reuse, and evolve to future requirements. Crosscutting works against modularization. Code for crosscutting concerns finds itself scattered through multiple modules; changes to that code now require changing all the places it touches, and (perhaps more importantly and less obviously) all changes to the system must conform to the requirements of the crosscutting concern. That is, if certain actions require, say, a security or accounting action, then in maintaining the code, we must consider how every change interacts with security and accounting.

Aspects. An *aspect* is a modular unit designed to implement a concern. An aspect definition may contain some code (or *advice,* which follows) and the instructions on where, when, and how to invoke it. Depending on the aspect language, aspects can be constructed hierarchically, and the language may provide separate mechanisms for defining an aspect and specifying its interaction with an underlying system.

Join points. *Join points* are well-defined places in the structure or execution flow of a program where additional behavior can be attached. A join point model (the kinds of joint points allowed) provides the common frame of reference to enable the definition of the structure of aspects. The most common elements of a join point model are method calls, though aspect languages have also defined join points for a variety of other circumstances, including field definition, access, and modification, exceptions, and execution events and states. For example, if an AOP language has method calls in its join point model, a programmer may designate additional code to be run on particular method calls.

Advice. *Advice* is the behavior to execute at a join point. For example, this might be the security code to do authentication and access control. Many aspect languages provide mechanisms to run advice *before*, *after*, *instead of,* or *around* join points of interest. Advice is *oblivious* in that there is no explicit notation at the joint point that the advice is to be run here—the programmer of the original base code may be oblivious to the evolving requirements. This contrasts with conventional programming languages, where the most common concern modularization mechanism, the subprogram, must be explicitly called.

Pointcut designator. A *pointcut designator* describes a set of join points. This is an important feature of AOP because it provides a *quantification mechanism*—a way to talk about doing something at many places in a program with a single statement. A programmer may designate all the join points in a program where, for example, a security code should be invoked. This eliminates the need to refer to each join point explicitly and thereby reduces the likelihood that any aspect code would be incorrectly invoked.

Composition. Abstractly, the idea of bringing together separately created software elements is *composition*. Different languages provide a variety of composition techniques, including subprogram invocation, inheritance, and generic instantiation. An important software engineering issue in the composition of components is the guarantees and mechanisms that a language provides to make sure that elements being composed "fit together." This allows warning of incompatibilities during system development, rather than being surprised by them during system execution. Common mechanisms for such guarantees include type checking the signatures of subprogram calls and the interface mechanism of languages like Java.

Weaving. *Weaving* is the process of composing core functionality modules with aspects, thereby yielding a working system. Various AOP languages have defined several mechanisms for weaving, including statically compiling the advice together with base code, dynamically inserting aspects when loading code, and modifying the system interpreter to execute aspects.

Wrapping, before and after. One of the most common AOP techniques is to provide method calls as (sometimes the only) join points and to allow the advice to run either *before*, *after,* or *around* the method call. This notion can be generalized into the idea of *wrapping*—providing a filter or container around a component, which mediates communications to that component and enforces the desired aspects.

Statics and dynamics. The terms *static* and *dynamic* appear in some of the following discussions. In general, *static* elements are ones that can be determined before the program begins execution, typically at compile time; *dynamic* things happen at execution. A weaving process can be either static or dynamic, depending on whether it relies on a compilation or loading mechanism (static) or run-time monitoring (dynamic) for its realization. Somewhat orthogonally, an AOP language can be characterized as having static or dynamic join points, depending on whether the places that aspects are to be invoked are dependent purely on the compile-time structure of the original code or the run-time events of program execution.

The tyranny of the dominant decomposition. The previous discussion spoke of aspects as something to be imposed on a "base" program. However, one can make a perfectly coherent argument that all code elements should be treated as equals and that the best way to build AOSD systems is by providing a language for weaving together such elements. Some of the systems discussed in this book adopt that point of view. A key subtext of that discussion is whether aspect behavior can be imposed on aspects themselves. Doing so makes the contractual assertions of aspects more complete at the cost of complicating the underlying implementations.

1.3 HISTORICAL CONTEXT

The history of programming has been a slow and steady climb from the depths of direct manipulation of the underlying machines to linguistic structures for expressing higher-level abstractions. Progress in programming languages and design methods has always been driven by the invention of structures that provide additional modularity. Subroutines assembled the behavior of unstructured machine instructions, structured programming argued for semantic meaning for these subroutines, abstract data types recognized the unity of data and behavior, and object-orientation (OO) generalized this to multiplicity of related data and behaviors.

The current state-of-the-art in programming is object-oriented (OO) technology. With objects, the programmer is supposed to think of the universe as a set of instances of particular classes that provide methods, expressed as imperative programs, to describe the behavior of all the objects of a class.

Object-orientation has many virtues, particularly in comparison to its predecessors. Objects provide modularization. The notion of sending messages to objects helps concentrate the programmer's thinking and aids understanding code. Inheritance mechanisms in object systems provide a way both to ascribe related behaviors to multiple classes and to make exceptions to that prescription.

Objects are not the last word in programming organization. This book is about an emerging candidate for the next step in this progression, aspect-oriented software development. Aspects introduce new linguistic mechanisms to modularize the implementation of concerns. Each of the earlier steps (with the minor exception of multiple inheritance in OO systems) focused on centralizing on a primary concern. AO, like its predecessors, is about recognizing that software systems are built with respect to many concerns and that programming languages, environments, and methodologies must support modularization mechanisms that honor these concurrent concerns. AO is technology for extending the kinds of concerns that can be separately and efficiently modularized.

Chapter 27

Developing Secure Applications Through Aspect-Oriented Programming

BART DE WIN, WOUTER JOOSEN, AND FRANK PIESSENS

In this chapter, we report upon our experiences using AspectJ to secure application software in a manageable way. Our case studies illustrate the effectiveness of AOP technology and show encouraging results. However, we also highlight some challenges to be addressed in the further development of aspect-oriented software development technology.

27.1 INTRODUCTION

Invariably, developing a real-life application demands that we consider both functional (i.e., related to the application business logic) and non-functional requirements. Separating the development of different requirements has important advantages in system evolution: As such requirements originate from different concerns (and very often from different stake-holders), they may cause different reiterations over various parts of the software life cycle. Successful separation of concerns thus leads to ease of development, maintenance, and potential reuse. State-of-the-art software techniques already support separating concerns, for instance, by using method structuring, clean object-oriented programming, and design patterns. However, these techniques are insufficient for more complex modularization problems. A major cause for this limitation is the inherently forced focus of these techniques on one view of the problem; they lack the ability to approach the problem from different viewpoints simultaneously. The net result is that conventional modularization techniques are unable to fully separate *crosscutting concerns*.

Aspect-oriented programming (AOP) is an approach that provides more advanced modularization techniques. The main characteristic of this technology is the ability to specify both the behavior of one individual concern and the way this behavior is related to other concerns (the *binding*). In fact, AOP has become a general term to denote several approaches to providing such development functionality. One prominent tool in this space is AspectJ [19]. AspectJ extends Java with mechanisms for expressing advanced modularization capabilities. In AspectJ, a unit of modularization is called an *aspect,* and a unit of binding is a *pointcut*.

This chapter reports the experience of developing security solutions for application software using AspectJ. We highlight both the advantages of aspect technology and remaining open challenges. The chapter is structured as follows: Section 27.2 briefly introduces the domain of application security. Section 27.3 covers two of the several case studies that we have developed. Section 27.4 evaluates AOP technology on the basis of our experience. We compare our work with alternative approaches to engineering application-specific security in Section 27.5, and we conclude in Section 27.6.

27.2 THE DOMAIN OF APPLICATION-LEVEL SECURITY

A classical and increasingly popular example of a non-functional concern is security. Security is a broad domain; we focus our research on the engineering of application-level[1] security requirements including authentication, auditing, authorization, confidentiality, integrity and non-repudiation. Security is a challenging application domain, particularly since many security experts are uneasy about trying to isolate security-related concerns. The primary reason for this discomfort is that security is a pervasive concern in software systems. Indeed, separating security-related concerns such as access control is difficult to achieve with state-of-the-art software engineering techniques.

A major cause of this pervasiveness is the *structural difference* between application logic and security logic. For example, the code to write relevant events to an audit log or the code that realizes an access control model is often spread among many classes. Attempts to modularize security concerns have been ongoing for many years. While the community has succeeded in modularizing the implementation of security mechanisms, *where* and *when* to call a given security mechanism in an application has not been adequately addressed. Furthermore, the crosscutting nature of security relates not only to the diversity of specific places where security mechanisms are to be called but also to the context of calls: Some security mechanisms require information that is not localized in the application. For instance, consider communication encryption within an application: the keys to be used for this purpose are typically linked to a user or principal that is somehow represented

1. As opposed to physical security and network layer security.

in the application. Key selection often depends on the specific communication channel and hence requires connection or host information. Initialization vectors and other security state information is often contained in the security mechanism itself. Finally, the actual data to protect may be scattered over several locations in the application.

Application-level security is an appealing but also a difficult candidate for validating AOP techniques because of its inherent complexity. In this validation, it is important to assess the flexibility for reuse and maintenance that a proposed mechanism provides. Given the prevailing heterogeneity of application domains and environments, there is a clear need to reuse security solutions. The extra importance of maintenance may require some further explanation. The Common Criteria [6] and their predecessors argue for considering security from the start of the system development process. However, history shows that for systems of moderate to high complexity, the idea of building a secure system from scratch is utopian. Unanticipated threats always arise during the lifetime of the system, both because the initial threat analysis was incomplete and because the environment in which the software operates changes. Some form of patching or updating the system is always necessary. Moreover, building a very secure system from the start makes an application complex and expensive, often beyond the economic resources of the developing organization.

27.3 AN EXPERIENCE REPORT

The goal of this section is to describe how AOP can be used to implement application security. We describe two case studies of security aspects. The first is a didactical example that illustrates the approach; the second applies that approach to a real-life application, a server for file transfer. We conclude with a discussion that generalizes these ideas in a reusable security aspect framework.

27.3.1 A Personal Information Management System

Our first example describes a Personal Information Management (PIM) system. A PIM system backs up the human memory by keeping track of personal information, including a person's agenda, contact information of friends and business contacts, the tasks he has to fulfill, and so forth. A Palm Pilot is a PIM. In this case study, we focus on the important requirement of access control.

Figure 27-1 shows the class diagram of a simplified PIM system. `PIMSystem` is the heart of the model. Through this class, the system can represent and manage three different types of information (or PIMUnits): appointments, contacts, and tasks. Besides a common operation represented in the abstract PIMUnit class, each information type requires different fields and operations. Finally, different Persons may perform operations on the system. Implementing access control in this system requires defining both the access control model and the mechanism for enforcing it. In our example, the *owner-based access control model* has the following rules:

- The owner (i.e., creator) of a PIMUnit can invoke all operations on that unit.
- Contacts are only accessible to their owner.
- All other accesses to PIMUnits are restricted to just viewing.

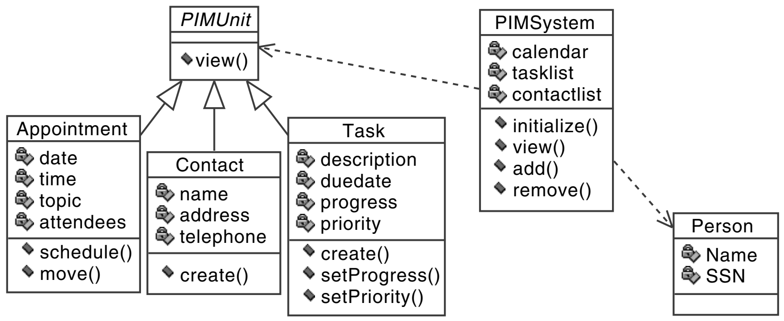

Figure 27-1 *PIM system class diagram.*

These rules are not complex. However, an object-oriented implementation of this access control model into the PIM context is not straightforward. First, every PIMUnit must be associated with its owner. This can be achieved by inserting an owner attribute into the PIMUnit class, initialized when the unit is created. Then, since access control requires that the real identity of the person responsible for initiating an operation is known, an authentication mechanism must be added to the model. We chose to do this in the general PIMSystem class. Finally, for the actual authorization checks, most operations in the four unit classes must be modified: The signature of the operations must be altered to pass identity information from PIMSystem to the authorization checks. As a result, the initial model has to be changed into a model, as shown in Figure 27-2, where the items in bold represent the places that require changes (both structural and behavioral). Crosscutting is epidemic.

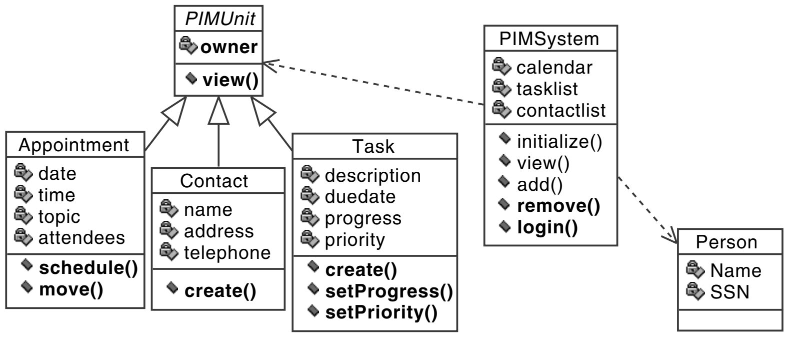

Figure 27-2 *PIM system modifications for access control.*

Listing 27-1 shows the implementation of the same access control functionality using AspectJ. The first aspect, `Ownermanagement`, is responsible for storage and initialization of the PIM unit owners. To this end, an instance of the aspect is associated (created) with every `PIMUnit` object. Every unit will be decorated with an owner attribute that is initialized by the *after* advice when the unit is created. The `Authentication` aspect is used to authenticate persons and replaces the `login()` method that was introduced in the object-oriented implementation of access control. An attribute is included in this aspect that represents the current user, and an operation initializes this attribute.[2] Finally, the `Authorization` aspect implements the actual access control. Here, an `around` advice verifies the equality of the owner of the unit that is currently being accessed and the current user as identified through authentication and acts accordingly. In this aspect, the `restrictedAccess` pointcut specifies the places for enforcing this verification.

The aspect approach as proposed previously could be considered equivalent to a regular object-oriented implementation, but at first sight it might seem more complex and thus less attractive. However, the integral modularization as described here has a number of essential advantages. First, coping with changes, especially unanticipated ones, is easier because all relevant code is gathered into one place. Second, the proper modularization simplifies scaling when the size of the application increases. This example is not extensive enough to demonstrate this, but for real-life systems,

2. For this aspect, several implementation scenarios are possible. In this example, we chose to support only non-simultaneous system interaction by making the currentUser static and thus system-wide. Other strategies, such as simultaneous access of persons, are discussed in [11]. The chosen strategy could also be implemented by associating the aspect with a `perthis()` statement to the `PIMSystem` class.

the difference will be considerable. Third, because of the proper modularization, developers can concentrate on the real core of the problem without having to worry about side issues such as code consistency. Finally, the complete separation of concerns improves the understanding of which security measures are implemented and where and when they are activated. These advantages will be clarified in the following paragraphs.

Listing 27-1 *Access control aspect implementation*

```
aspect OwnerManagement perthis(this(PIMUnit)){
  String owner ; //one per PIMUnit object
  after(): execution(Appointment.schedule(..)) ||
           execution(Contact.create(..)) ||
           execution(Task.create(..)){
    owner = Authentication.getUser() ;
  }
}

aspect Authentication(){
  static String currentUser ; //one per system
  static String getUser(){
    if(currentUser == null) currentUser = <login> ;
    return currentUser ;
  }
}

aspect Authorization(){
  pointcut restrictedAccess():
    execution(* Appointment.move(..)) ||
    execution(* Contact.view(..)) ||
    execution(* Task.setProgress(..)) ||
    execution(* Task.setPriority(..)) ;

  void around() : restrictedAccess(){
    Object currentUnit = thisJoinPoint.getThis() ;
    String unitOwner = OwnerManagement.aspectOf(currentUnit).owner;
    String user = Authentication.getUser() ;
    if(! unitOwner.equals(user))
      System.out.println("Access Denied !") ;
     else proceed() ;
  }
}
```

Consider a possible evolution of the PIM system. Suppose it is used by a company where different participating roles are defined, such as secretaries, managers, and so forth. The original design needs to be changed to reflect the different roles (which can be done by subclassing `Person` with several classes). In such an environment, the access control subsystem might also require an update. For example, secretaries may require full access to all information of their managers. This new requirement affects all access control checks. In the object-oriented security solution, these operations are dispersed among several classes (`PIMUnit`, `Appointment`, `Task`, and `Contact`). This extension requires considerable effort by the software maintainer, along with the possibility of introducing maintenance mistakes. For our aspect implementation, however, the only required change is the modification of the condition of the "if-statement" within the `Authorization` advice so that it includes this new access rule. The changes required to support this extension are more localized.

This example of changing the access control model clearly demonstrates the flexibility gained by the advanced modularization capabilities of aspect-oriented programming. Similarly, other models can be supported equally elegantly. For instance, an ACL (Access Control List) based model would allow the owner to define fine-grained rules for the access rights of each person. In addition, different information confidentiality levels (such as public, confidential, and top-secret) could extend the access control model in order to further restrict information access in the system based on the user's clearance level. The AOP approach can even support a capability-based model, where one can delegate access privileges to others. All these models can be supported without requiring invasive changes in the core application software or unwarranted changes in the aspect implementation.

27.3.2 An FTP Server

Our second case study deals with the security requirements of jFTPd [17], a server that (partially) implements the well-known File Transfer Protocol (FTP). The implementation includes several security measures, most of which are imposed by the specification of the protocol. This case study only discusses access control, which is user-based. Users authenticate with a (user name and) password, once per connection, and then the current connection is linked with this user. FTP commands are executed only after proper authorization.

In the implementation of the FTP, `FTPHandler` is the central class in the model that takes care of incoming connection setup requests. For every request, `FTPHandler` instantiates an `FTPConnection` object and assigns the incoming connection to it. From this point onwards, the latter acts as the primary contact point for the connection, which means that it is responsible for reading and answering all incoming FTP requests on the connection. It has a central input operation that delegates each command input to an appropriate suboperation depending on the specific request. At the end of an FTP session, the connection is closed, and the connection object is destroyed.

A user-based access control model can be successfully implemented using AOP technology. We briefly sketch the aspect structure and strategy used.[3] FTP security is session-oriented—the credentials presented at login are used throughout the entire session. Therefore, similar to the `OwnerManagement` aspect of the previous example, an `FTPSession` aspect is associated with the `FTPConnection` class and holds the information from the authentication phase. Furthermore, a second aspect, `FTPConnectionSecurity`, performs two important tasks: In the authentication phase, it checks username/password combinations (and fills in the outcome in the `FTPSession` aspect), and in the various FTP commands of the `FTPConnection` class, it performs the actual access control. Finally, the aspect implementation includes two other aspects for dealing with the representation and initialization of some other security parameters that are not relevant to this discussion. Through the aspect implementation, a complete separation of security-related code is achieved from the initial implementation of the FTP server. In other words, we are able to deploy the FTP server with or without weaving in (i.e., bind and activate) the security aspects.

27.3.3 Toward a Framework of Aspects

The deployment of the aspects described in the previous examples depends heavily on the type and implementation of the actual application. For instance, the `Authorization` aspect of Section 27.1 can only be used for `PIMUnit` classes, not for `FTPConnections`. Also, the explicit choice to have system-wide personal authentication in the first case study limits the applicability of that specific aspect. In general, it seems hard to define a single set of aspects that is applicable in a broad range of applications. However, just as Java classes can be made abstract to represent generic behavior, one would like the ability to implement generic aspects that can be reused in several cases. A closer look at the previous aspects reveals two obstacles that hinder their reuse.

- First, the design of (and the mechanisms used within) the specific aspects differs among different cases. An illustration of this is the fact that the number of aspects differs in the two case studies presented in this chapter, although they both cover access control.
- The second obstacle is related to the explicit deployment statements in the definition of an aspect. As an example, aspects that crosscut with `PIMUnits` cannot be applied to `FTPConnections`.

3. A more elaborate discussion on this case study is presented in [8].

The key to resolving the first problem is identifying the generic domain concepts and their mutual dependencies while ignoring the particular implementation details of the underlying application. In the two case studies, three common concepts can be identified: (1) an authentication concept that stores information about the subject initiating an access attempt and that guarantees the correct identity of the subject, (2) a resource concept that models resource-specific information required for access control, and (3) an authorization concept that implements the access controls for every attempt to access the resource, possibly based on subject and resource information. In the first case study, the three aspects map nicely onto the previous concepts (notice that `OwnerManagement` implements the resource concept). In the second case study, both discussed aspects (`FTPSession` and `FTPConnectionSecurity`) actually represent distinct parts of the authentication concept and hence should be merged. The authorization concept maps directly; the resource concept has not been used. In short, it is fair to state that the technology does not stop us from generating reusable results.

To address the second problem, AspectJ supports abstract definitions of pointcuts. Specific pointcuts can be instantiated afterwards in an extended aspect. Using this mechanism, it is possible to build a general aspect and redefine the abstract pointcuts based on a specific application.

Combining the two solutions discussed here results in aspect code as shown in Listing 27-2. While the intellectual effort of this generalization phase is nontrivial, it allows the reuse of the core structure of the security aspects. A qualified person properly designs security solutions once. Later, aspect inheritance enables reuse. Furthermore, by continuing this exercise for other security requirements (such as confidentiality, non-repudiation, etc.), a combination of security aspects can be built that form the basis of an aspect framework for security. This framework consists of core structures modeling security requirements, concrete mechanism implementations for these requirements, and abstract pointcuts that must be extended for specific applications. In Listing 27-2, the concrete mechanisms still require an implementation at the places represented by "<. . .>". A more elaborate discussion on this security framework can be found in [31].

Listing 27-2 *Generalized aspects for access control*

```
abstract aspect Authentication perthis(entities){
  abstract pointcut entities() ;
  abstract pointcut authenticationPlace() ;
  private String id ;
  after(): authenticationPlace(){
    id = <authenticate user> ;
  }
}
```

```
abstract aspect ResourceInformation perthis(resources) {
  abstract pointcut resources ;
  ...
}

abstract aspect Authorization{
  abstract pointcut serviceRequest() ;
  void around(): serviceRequest(){
    <check access>
  }
}
```

27.4 DISCUSSION

27.4.1 A Positive Experience

Optimizing the separation between application and security logic is an important objective. Our experience with aspect-oriented programming confirms that this technology enables us to achieve this goal. Several smaller case studies made us confident about the potential of this technology, but the actual implementation of the FTP server lifted our confidence. This is not a toy application. Nevertheless, we were able to fully extract security-related code, producing an isolated basic application. This was done despite the fact that the FTP server was not our own code.

AOP has two important advantages for security. Compared to the well-known technique of modularizing the security mechanism, AOP allows us to raise the level of separation by explicitly focusing on the binding between application and security infrastructure. As a result, the average application developer is no longer involved with security (i.e., he no longer has to invoke security mechanisms himself). This job can be left completely to a focused security expert. A second key advantage relates to the security policy rather than to the infrastructure. Using AOP, the overview of the actual security deployment policy (i.e., defining which mechanisms are used and where they are used) is gathered into a few configuration files. Consequently, compared to object-oriented security engineering, a security expert can more easily verify whether a required security policy is valid within a concrete application.

Some practical limitations of the current tools limit the success of our approach. During different case studies, we experienced some technological restrictions with (version 1.1 of) AspectJ, including limitations of the mechanism to select join points, as well as some essential restrictions in the generalization phase toward the framework. To briefly sketch the first problem, a pointcut is conceptually equivalent to a query on an abstract syntax tree representation of the program. The AspectJ tool provides a set of keywords to describe pointcuts. Unfortunately, some queries (e.g., all the classes that override one or more methods of their parent classes) cannot be expressed using this keyword set, which leads to the construction of less elegant workarounds. Regarding the second problem, aspects, and in particular pointcuts, can be made abstract, but they can only be reused in child aspects of the abstract aspect in which they were defined. In our opinion, the primary cause of this restriction is the forced combination of the specification of behavior and composition logic. Since AspectJ aspects are not fully polymorphic, aspect reuse is hard to achieve. We refer to [10, 12, 31] for a more in-depth discussion of these problems. Some of the restrictions we encountered (e.g., the first one) are due to the current implementation of the tool, while others (the second one) are more fundamental issues that are inherent to the basic concepts behind the tool. As AOP is an active research area, this is what one should expect. Case studies, as included in this chapter, should drive the environments to the next stage. Instead of focusing on specifics related to AspectJ, we will focus in the next paragraphs on the requirements for the AOP domain as a whole.

27.4.2 Requirements for AOP Environments

Define the optimal design process. AOP is a new programming paradigm building on established paradigms such as object-oriented programming. Unfortunately, this hinders the average programmer trying to become productive. Conceptually clean and understandable design processes can help alleviate this problem. Obviously, the rapidly changing character of the AOP technology as it is today does not simplify the development of such clean processes. Even with the existence of such processes, the implementation of security aspects still requires detailed knowledge of the security infrastructure. Apart from enabling improved modularization, AOP technology does not help here.

Watch performance. With AOP, the modules of a program are composed into an executable artifact. Many tools currently transform an aspect-based program into a classical (class-based) object-oriented program. This results in extra methods and extra method invocations relative to the equivalent handcrafted program. Moreover, execution of a transformed program often requires extra run-time libraries. Development and language support often involve a trade-off between efficiency and ease of use. Historically, the transition from procedural to object-oriented programming involved the same trade-off. The important issue here is that these languages or tools disable some of the optimization capabilities of the programmer and replace them with a more expressive programming model. While we did not experience unacceptable penalties when testing and deploying the aspect-based FTP server, we stress the relevance of this subject matter in the long run, as technology and tools mature.

For AspectJ in particular, the definition of advice on security-sensitive methods results in the insertion by the AspectJ compiler of one or more extra calls. For instance, a before advice is implemented using a method proxy, which requires extra indirection. Therefore, the generated code is less efficient and introduces more overhead than a direct implementation. Unfortunately, this is the price of generality. Building a less general aspect and a more complex combination tool could improve this situation.

Support testing and debugging. Debugging involves code assessment and correction. In a typical design process, all sorts of tests (black box, white box, stress, and so forth) are used to validate modules. Likewise, when using AOP, the programmer should have the ability to test aspects. This testing must include both the behavior of the aspect, which is part of the security infrastructure, and the binding within different environments (the security policy). Testing aspect behavior is comparable to but different from traditional object testing. Testing the binding is difficult. It must include both information about the deployment environment (type information and possibly context information) and specific application binding information (specific state transferred between different aspects). Today, research on aspect testing is in a very early state, and practical tools are non-existent. Locating erroneous code is complicated by the fact that the running code does not correspond to the written code. Fortunately, current AOP research also focuses on tools that provide a clear visualization of the run-time interaction between aspects and the core application.

Toward trusted code. From a security viewpoint, the explicit separation and consequent composition of security aspects and application objects raise the extra risk of introducing new security holes. In particular, the specific combination mechanisms used in the tool and the tool itself must be part of the trusted computing base, and therefore, they are a primary target for attacks. Let us consider the case of authorization: It should not be possible for a client to directly invoke the end-functionality of a server, circumventing the authorization policy. In this respect, security demands absolute guarantees that the combination process (a.k.a. weaving) cannot be bypassed. At the moment, it is not clear how such guarantees can or will be enforced. This is an important topic for further research.

Specifically, in the context of AspectJ, the output of the actual tool cannot be trusted because the original functionality, without the new aspect code, is simply moved into a new method with a special name. Clever attackers knowing the modus operandi of the tool can exploit the code very easily. As explicitly stated by the tool manual, this problem only arises if not all source code is under the control of the

AspectJ compiler. Unfortunately, for practical real-world situations, such as large development processes or dynamic modification of an application, this requirement is often impossible to fulfill.

27.5 RELATED WORK

We first consider alternative ways to add security to an application. Broadly speaking, one can distinguish between the use of component libraries for security and other work that focuses on the security binding.

Over the past few years, several security libraries have been developed to enable the implementation of application-specific security requirements. For example, Sun has released JCA/JCE with basic security primitives, JAAS [20] for authentication and authorization, and JSSE for secured network communication. Other examples include Pluggable Authentication Modules (PAM) [26] and GSS API [21]. We consider them as component libraries: They successfully capture the domain logic into separate modules, enabling flexible component selection and interchange. Unfortunately, the use of component libraries does not improve the control over the inherent crosscutting nature of security.

Ongoing research (e.g., Naccio [13], Ariel [24], and PolicyMaker [3]) addresses a declarative description of security properties for application software. Such security-specific language would be especially valuable to define the binding between application and security mechanisms. In this work, the core challenge is to create the right security abstractions once and for all. We have chosen to focus on a general-purpose aspect-oriented language, and we believe that security abstractions will remain an evolving challenge for the foreseeable future.

Next, we consider related work from a different angle by discussing alternative software engineering techniques to deal with crosscutting concerns. From the viewpoint of software architectures, several approaches have addressed increased independence between application logic and security logic.

By using a number of object-oriented design patterns [16], many security architectures try to be independent of an application structure. This can be achieved to some extent. Also, various security-specific patterns have been proposed [27]. The drawback of this approach is that the structure of the solution becomes more complex and harder to understand. This additional complexity is hard to accept in light of security validation requirements.

Meta-level architectures [4, 25, 29] enable the separate implementation of application and security functionality [1, 33]. They offer a complete reification of the execution of the application: The events of sending a message, starting the execution, or creating objects all get reified into first class entities. As the meta-program has control over these reified entities, it can affect the execution of the core application. In comparison to aspect-oriented programming, this mechanism is in many ways more powerful, but it is also a much heavier burden for the software engineer. Moreover, the development of meta-programs for security is more complex because the programmer is forced to think in terms of meta-entities only indirectly related to the application. Like security patterns, meta-level architectures complicate the security validation process.

Other approaches, such as the CORBA Security Service [2, 23], Microsoft .NET [22], JBoss [30], and others [9, 14, 15], use the basic idea of introducing an interceptor between clients and services, in order to perform access control, for example. They are similar to meta-level architectures in that they intervene in the communication between client and service, but the intervention is less generic (and as such more simplified): the interceptors are mere decorators to the services. In straightforward situations, they can be specified fairly easily, possibly through declarative description. However, when more and more application state needs to be taken into account, writing decorators becomes extremely hard.

Transformations in AspectJ happen on the level of source code, and similar results were achieved in AOSF [28, 32]. Other tools are available that work on the level of bytecode [5, 18]. This has the advantage that aspects can be added even when no source code is available for the application. The disadvantage is that on the level of bytecode, a lot of the application logic is already lost. Reconstructing this logic is hard, and producing correct descriptions of how a series of bytecodes has to be changed to implement authentication is even harder. Needless to say, validating and debugging the result will be challenging. We believe that with the current state-of-the-art, an approach that manipulates source code is important to enable the rapid evolution of various stages in the technology. This will be important in order to experiment with AOP environments of growing maturity, while capturing new requirements as experience grows.

27.6 CONCLUSION

In this chapter, we have discussed our experiences in using aspect-oriented programming to develop security components for distributed applications. We believe that we have illustrated the effectiveness of AOP technology with two application-level security problems, which resulted in examples that are beyond toy-level demonstrations of the technology at hand. In the long run, however, it is clear that support at the level of development processes and environments will be essential for aspect-based technology to become widespread.

We believe that security greatly benefits from enhanced modularization with aspects. The key advantages in this context are the full separation of business and security logic, which allows security experts to concentrate on their core business, and the centralization of the security policy that raises policy verification to a higher level. A recent doctoral thesis [7] discusses the feasibility, merits, and drawbacks of using aspect-oriented software development for application-level security in more detail.

Finally, it is also fair to state that by *only* focusing on security, we have not covered the challenge of combining aspects that result from complementary, relatively unrelated concerns. The use of the technology in such a context, where advanced aspect compositions is clearly required, is the focus of ongoing and future work.

ACKNOWLEDGMENTS

The authors would like to thank Eddy Truyen, Bart De Decker, and the other members of the DistriNet research group for the interesting and inspiring discussions in the context of this work. Furthermore, the Department of Computer Science at the K. U. Leuven deserves our gratitude for giving us the opportunity to conduct this research. Finally, we would like to thank the editors for their useful comments and suggestions for improvements.

REFERENCES

1. Ancona, M., Cazzola, W., and Fernandez, E. B. 1999. Reflective authorization systems: Possibilities, benefits, and drawbacks. In *Secure Internet programming: Security Issues for Mobile and Distributed Objects*, J. Vitek and C. Jensen, Eds. LNCS, vol. 1603. Springer-Verlag, Berlin, 35–49.

2. Beznosov, K. 2000. Engineering access control for distributed enterprise applications. Ph.D. thesis, Florida International University, Miami, Florida.

3. Blaze, M., Feigenbaum, J., and Lacy, J. 1996. Decentralized trust management. In *1996 Symp. Security and Privacy,* (Oakland, California). IEEE, 164–173.

4. Chiba, S. 1995. A metaobject protocol for C++. In *10th Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, (Austin). ACM, 285–299.

5. Cohen, G., Chase, J., and Kaminsky, D. 1998. Automatic program transformation with JOIE. In *1998 Annual Technical Symposium* (New Orleans). USENIX, 167–178.

6. Common Criteria. 1999. Common criteria for information technology security evaluation, version 2.1. Tech. rep., Common Criteria. http://www.commoncriteria.org.

7. De Win, B. 2004. Engineering Application-level Security through Aspect-Oriented Software Development. Ph.D. thesis, Katholieke Universiteit Leuven, The Netherlands.

8. De Win, B., Joosen, W., and Piessens, F. 2003. AOSD & security: A practical assessment. In *Software Engineering Properties of Languages for Aspect Technologies (SPLAT)*, (Boston). http://www.daimi.au.dk/~eernst/splat03/papers/Bart_De_Win.pdf.

9. De Win, B., van den Bergh, J., Matthijs, F., De Decker, B., and Joosen, W. 2000. A security architecture for electronic commerce applications. In *Information Security for Global Information Infrastructures*, S. Qing and J. Eloff, Eds. Kluwer Academic Publishers, Boston, 491–500.

10. De Win, B., Vanhaute, B., and De Decker, B. 2001. Towards an open weaving process. In *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA)*, (Tampa, Florida). http://www.cs.ubc.ca/~kdvolder/Workshops/OOPSLA2001/submissions/07-dewin.pdf.

11. De Win, B., Vanhaute, B., and De Decker, B. 2002. How aspect-oriented programming can help to build secure software. *Informatica 26*, 2, 141–149.

12. Ernst, E. and Lorenz, D. H. 2003. Aspects and polymorphism in AspectJ. In *2nd Int'l Conf. Aspect-Oriented Software Development (AOSD)*, (Boston), M. Akşit, Ed. ACM, 150–157.

13. Evans, D. and Twyman, A. 1999. Flexible policy-directed code safety. In *Symp. Security and Privacy*, (Oakland, California). IEEE, 32–45.

14. Filman, R. E., Barrett, S., Lee, D. D., and Linden, T. 2002. Inserting ilities by controlling communications. *Comm. ACM 45*, 1 (Jan.), 116–122.

15. Fraser, T., Badger, L., and Feldman, M. 1999. Hardening COTS software with generic software wrappers. In *Symp. Security and Privacy*, (Oakland, California). IEEE, 2–16.

16. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts.

17. jFTPd. jftpd, ftp server with remote administration. http://homepages.wmich.edu/~p1bijjam/cs555 Project/.

18. KELLER, R. AND HÖLZLE, U. 1998. Binary code adaptation. In *ECOOP'98 Object-Oriented Programming, 12th European Conference*, E. Jul, Ed. LNCS, vol. 1445. Springer-Verlag, Berlin, 307–329.

19. KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *ECOOP'97 Object-Oriented Programming, 11th European Conference*, M. Akşit and S. Matsuoka, Eds. LNCS, vol. 1241. Springer-Verlag, Berlin, 220–242.

20. LAI, C., GONG, L., NADALIN, A., AND SCHEMERS, R. 1999. User authentication and authorization in the Java platform. In *15th Annual Computer Security Applications Conference*, (Phoenix, Arizona). IEEE, 285–290.

21. LINN, J. 1997. RFC2078: Generic security service application program interface, version 2. Tech. rep., IETF. http://www.ietf.org/rfc/rfc2078.txt.

22. LOWY, J. 2003. Decoupling Components by Injecting Custom Services into your Object's Interceptor Chain. In *MSDN Magazine 03/03*.

23. OBJECT MANAGEMENT GROUP. 2002. CORBA security service specification, version 1.8. http://www.omg.org.

24. PANDEY, R. AND HASHII, B. 1999. Providing Fine-Grained Access Control for Java Programs. In *ECOOP'99 Object-Oriented Programming, 13th European Conference*, R. Guerraaoui, Ed. LNCS, vol. 1628. Springer-Verlag, Berlin, 449–473.

25. ROBBEN, B., VANHAUTE, B., JOOSEN, W., AND VERBAETEN., P. 1999. Non-functional policies. In *Meta-Level Architectures and Reflection*, P. Cointe, Ed. LNCS, vol. 1616. Springer-Verlag, Berlin, 74–92.

26. SAMAR, V. AND LAI, C. 2003. Making login services independent of authentication technologies. Tech. rep., Sun Microsystems, Inc. http://java.sun.com/security/jaas/doc/pam.html.

27. SECURITY PATTERNS HOME PAGE. http://www.securitypatterns.org/.

28. SHAH, V. AND HILL, F. 2004. An Aspect-Oriented Security Framework: Lessons Learned. In *AOSD Technology for Application-level Security (AOSDSEC)*, (Lancaster). http://www.cs.kuleuven.ac.be/~distrinet/events/aosdsec/AOSDSEC04_Viren_Shah.pdf.

29. STROUD, R. AND WUE, Z. 1996. Using metaobject protocols to satisfy non-functional requirements. In *Advances in Object-Oriented Metalevel Architectures and Reflection*, C. Zimmermann, Ed. CRC Press, Boca Raton, Florida, 31–52.

30. TAYLOR, L. 2002. Customized EJB Security in JBoss. http://www.javaworld.com/javaworld/jw-02-2002/jw-0215-ejbsecurity.html.

31. VANHAUTE, B., DE WIN, B., AND DE DECKER, B. 2001. Building frameworks in AspectJ. In *Workshop on Advanced Separation of Concerns (ECOOP)*, (Budapest). http://trese.cs.utwente.nl/Workshops/ecoop01asoc/papers/ VanHaute.pdf.

32. VIEGA, J., BLOCH, J. T. AND CHANDRA, P. 2001. Applying Aspect-Oriented Programming to Security. In *Cutter IT Journal 14, 2*, 31–39.

33. WELCH, I. AND STROUD, R. 2003. Re-engineering Security as a Crosscutting Concern. In *The Computer Journal 46, 5*, 578–589.