

On Designing Access Control Aspects for Web Applications

Kung Chen and Chih-Mao Huang
Department of Computer Science
National Chengchi University
Wenshan Dist., Taipei 106,
Taiwan
{[chenk, g9224](mailto:chenk, g9224@cs.nccu.edu.tw)}@cs.nccu.edu.tw}

ABSTRACT

This position paper reports our experience in designing access control aspects for Web applications. In particular, we choose the MVC-based Struts framework as the architectural style of our target applications. Two aspect suites in AspectJ are developed to enforce fine-grained access control in a modular and non-invasive manner. While both suites exhibit good software engineering properties, the first suite cannot guarantee the required conditional execution and the second suite achieves it at the cost of code duplication. We elaborate on the underlying issues of aspect composition and present our preliminary findings from this study.

Keywords

Access Control, Aspect-Oriented Programming, Framework, Web Applications.

1. INTRODUCTION

Application-level access control is a typical instance of crosscutting concerns that aspect-oriented programming (AOP) aim to modularize well. As demonstrated in the pioneering work of De Win [4][5], AOP is a promising approach to handle access control enforcement in a highly modular manner. Here we carry on this line of investigation further. Yet, instead of doing a single case study, we will look into a whole class of applications, namely, Web applications. A major goal of this study is to examine how good the mainstream architectural styles of Web applications can support the aspect-oriented approach to *fine-grained* access control enforcement. We intend to accommodate a wide range of fine-grained access control requirements while keeping the associated access control aspects as non-invasive as possible to the underlying application structures.

Fine-grained access control constraints usually involve, in addition to the designated function, the specific data contents of a user request. For example, in a B2B E-Commerce site, users from any registered organizations have the permission to execute the “viewOrder” function, but they are allowed to view only orders of

their own organization. This is called instance-level access control constraints [8]. Furthermore, within a data record, certain specific elements, such as credit card number, may have to be excluded from screen view to protect the user’s privacy. We refer to this as field-level access control constraints. These constraints are very common in Web-based E-Commerce applications, yet they have not been addressed in previous work based on AOP.

This position paper reports our experience and findings from this study. We approach our goal through a proper analysis of access control granularity and the architectural patterns of Web applications. The techniques of design patterns and template advice [9] are then employed to design the aspects. We present two aspect suites in AspectJ [2] that can enforce fine-grained access control in a modular and non-invasive manner. The first suite separates authentication aspects from other access control aspects, yet the second one integrates both and forms an aspect framework [17].

While both aspect suites exhibit good software engineering properties, we still encountered some issues of aspect composition during this study. Specifically, the aspects in the first suite have simpler structures, yet they cannot guarantee the required conditional execution. In contrast, those aspects in the second suite achieve guaranteed conditional execution at the cost of code duplication. This motivates us to further study the issues using other AOP tools. We choose the recently released JBoss AOP [11], which supports AOP in plain Java classes. The preliminary findings are not very positive, but they should shed some light on the strong and weak points of the aspect composition mechanisms provided in current AOP languages and tools.

The remainder of this paper is organized as follows. Section 2 set up the context of our work by presenting an analysis of approaches to access control. Section 3 discusses our design considerations and the architectural patterns we require from the underlying Web applications. Section 4 presents the access control aspects and explains why we developed an alternative implementation. Section 5 elaborates on our experience from this study and presents some preliminary findings on composing aspects using JBoss AOP. Finally, section 6 concludes.

2. CHARACTERIZATION OF ACCESS CONTROL APPROACHES

Access control, also known as authorization, is a process that determines whether an identified user has the privilege to access certain resources. The step to identify a user is usually called

authentication. These two security services are very closely related, for authorization cannot be accomplished without first performing authentication. Hence, in the following discussion, by access control we often include user authentication as part of the process.

Modern application platforms such as J2EE and MS.Net provide a high-level API that abstracts the underlying system-level access control mechanisms and allows application developers to separate access control enforcement from functional code to a certain degree. For example, the Java Authentication and Authorization Services (JAAS) [16] of J2EE provide a programmatic API for enforcing general access control as well as declarative security for conducting straightforward access control through a configuration file.

While sufficient for meeting many common access control requirements encountered in application development, APIs like JAAS cannot fully relieve the problem of code scattering and tangling. Specifically, it is very common that application-level authorization decisions must be based on factors specific to an application's state, other than simple user identity and roles. Here the declarative security of JAAS cannot help; we have to add calls to JAAS to all business logic where such decisions are needed. As the business logic is spread over multiple modules, so too is the implementation of the access control logic.

To further clarify the issues involved, we present an analysis of access control approaches from two different dimensions. One is granularity level which concerns user requirements; the other is implementation technology adopted by application developers. First, we model the interaction between a user and an application system as a sequence of three-tuples: $\langle user, function, data \rangle$, indicating a user's request to execute the function on a specific data object. The security administrator will specify access control rules with constraints that must be satisfied to grant any access requests based on the attributes of the three elements in the tuple. For example, the following rule demands that, after password authentication, only VIP customers are allowed to execute the `createOrder` function with total amount argument exceeding \$100,000:

```
<createOrder, AuthType=Password,
  (lessEq(Fun.getArgument("total"), 100000)
   || defined(User.getAttr("VIP")))>
```

Along the lines of thought we divide the granularity of a system's access control into three levels: *user*, *function*, and *data*. User-level granularity allows a user to access anything if he or she is a legitimate user, e.g., pass the password check. Function-level granularity restricts a legitimate user's access to application functions based on the attributes of the user and the function elements, yet regardless of the data contents being accessed. The VIP rule stated above belongs to this category. Data-level granularity is the most fine-grained one that also takes the contents of the data to be accessed into consideration when making the decision. For instance, the following rule dictates that customers can list only orders of their own.

```
<listOrders, AuthType=Password,
  (User.getRoles().contains("customer")
   && equals(User.getName(), Data.getOwner()))>
```

Second, we divide the implementation technologies for access control into three different levels: *hard-wired*, *adaptable*, and

configurable. Hard-wired implementation means that the code for enforcing access control is scattered through the system and mixed with other functional code. This is quite common in existing systems. In contrast, both adaptable and configurable implementations require that the access control code is properly modularized and can be adapted to new requirements with little efforts. The key difference between them is that a configurable implementation enables us to set the access control rules in a non-programming language, such as XML, and afterwards to revise only the rules to get a different access control setting; while in a highly adaptable implementation, we still need to look into the source code to make the necessary changes [18].

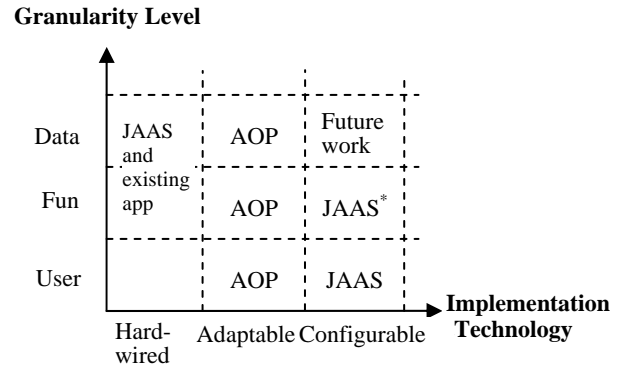


Figure 1: Comparing Access Control Approaches

By combining these two dimensions, we get nine different configurations of granularity and implementation technology for access control. These configurations span a wide spectrum that covers existing systems, security services of modern application platforms, emerging application development technology, and the ideal approach. Figure 1 shows the relative positions of representative approaches based on this framework. Many existing applications fall in the squares of hard-wired implementation, especially in the presence of fine-grained requirements. JAAS allows configurable implementations for simple user-level access control, and supports configurable function-level access control in a very limited context where user attributes such as roles are the only criteria for making the access control decisions¹. On the other hand, we will show that, in our study context, AOP can support a high degree of adaptability for all levels of access control. Furthermore, we claim that it is also feasible to derive a configurable implementation for fine-grained using AOP.

3. DESIGN RATIONALE

This section highlights the design rationale behind our access control aspects for Web applications.

3.1 Desiderata

Access control is a system-wide concern that cut across functional modules. Moreover, to make an access control decision, especially fine-grained ones, the aspect code need substantial information from the application about the context in which the decision is required. It is not something like function tracing that

¹ This is indicated by the asterisk attached to JAAS in Figure 1.

is completely orthogonal to functional modules and all one has to do is define the functions to be traced as the pointcuts without any involvement from the underlying function. In contrast, access control aspects depend on the cooperation from the application to a significant degree. We need to carefully look for the proper application pointcuts to weave in aspect code so that a suitable balance between information need and non-invasiveness is achieved.

This turns out to be more of a challenge. It is often the case that, in order to obtain the needed information, we must take some invasive measures to adapt an application. This seems to be unavoidable, yet we do not plan to invent any new architectural styles for the underlying applications. Instead, we decided to look into the mainstream architectural patterns for Web applications which have proven records and may suit our purpose.

In addition to following current architectural patterns, we set up three criteria for choosing the pointcuts for hooking access control aspects. First, all the information required for enforcing access control rules and filtering out unauthorized contents must be available to the aspect code, since fine-grained access control cannot be realized without detailed application state. Second, in case that the attempted access must be denied, exception propagation and handling must not incur significant impacts on the underlying program structure. The same requirement holds for filtering out unauthorized contents. Third, the correspondence between an access control rule and an enforcing aspect should be direct and clear for management and maintenance purposes.

3.2 Architectural Style Decision

A good starting point for structuring Web application is the three-tier architectural principle of dividing an application into three logical areas: presentation, business logic, and data processing. In the beginning, it is tempting to consider functions inside the business logic tier or the data tier as the proper hook points for the aspects. However, after some investigation, we reached the conclusion that they are not the ideal targets for achieving our objectives. It can easily lead to messy code for passing the required information or handle access denying exceptions. Neither does the internal logic of the presentation tier a good place for handling unauthorized data contents for similar reasons.

After some investigations, we choose the well-accepted Model-View-Controller (MVC) [7] architectural pattern as the structural guideline for our Web applications. In MVC, the model components encapsulate the application components in the business logic and data tiers. The view components are those pieces of an application that display the information provided by model components and accept input. The controller is a special program unit that coordinates all the activities of the model and view components. MVC is indeed a very good example of *separation of concerns*; it not only organizes the presentation tier well but also preserves a clean interface with the business tier.

Furthermore, many architectural frameworks for Web applications are based on MVC; this decision would greatly increase the applicability of our approach. Here we adopt the very popular Java-based Apache Struts framework [1] for our study. Figure 2 illustrates the architecture of Struts-based Web applications. Every user request is dispatched to an action class by the controller according to the action mapping defined in the

configuration file, *struts-config.xml*. These actions are responsible for serving user requests or passing them to the designated business tier components, and for returning the correct view element that the controller should forward to after finishing the user request. This view forwarding is also based on the mapping information specified in the configuration file.

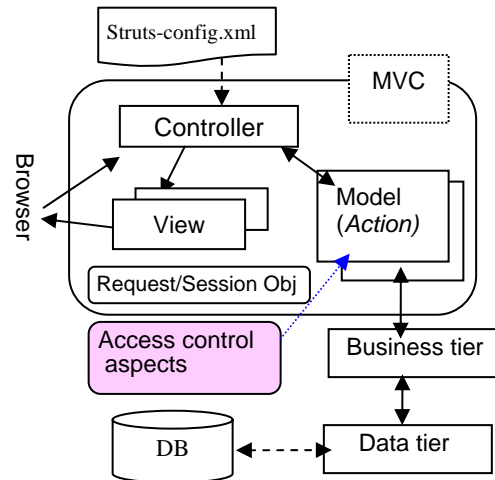


Figure 2: Struts-based Web Applications

In other words, the action classes play the role of gateway between the presentation tier and the business and the data tiers. Moreover, all action classes must inherit from the class *Action* and implement an *execute* method with the following signature.

```
public ActionForward execute
(ActionMapping mapping, ActionForm form,
HttpServletRequest req,
HttpServletResponse resp)
```

We find that the *execute* method is indeed the proper join point to weave in our advice code for its arguments expose the right information we need. First, user input and any intermediate results, including user authentication records and user profiles, are all stored in the request or session objects, which are available through the *HttpServletRequest* argument to the method. Therefore, via the *args()* pointcut, our aspects will have all the information needed for enforcing the access control and filtering out unauthorized contents.

Second, the control transfer between actions and views is specified in the *struts-config.xml* file, and is realized through the *action mapping* argument passed to the method and the *ActionForward* object returned by an action class. Hence we can define proper exception pages in the configuration file and forward to them when an access request is denied in the aspect code. Apparently, this exception handling scheme fits very well into Struts. Third, as will be shown later, an access control rule corresponds to its enforcement aspect quite directly since a user function will usually be supported by an action class.

4. ACCESS CONTROL ASPECTS

This section presents the details of the access control aspects we designed and implemented in AspectJ for Struts-based Web applications. Due to the lack of proper aspect composition

mechanisms, we developed an alternative implementation that achieves guaranteed conditional execution at the cost of code duplication.

4.1 Defining the Aspects

We define three abstract aspects that capture the common code structure for enforcing fine-grained access control, namely *Authentication*, *Precheck*, and *Postfilter*. All of them follow the pattern of template advice [9], and they have a rough correspondence to the three granularity levels described in Section 2, namely, user, function, and data.

4.1.1 Authentication Aspects

The authentication aspect is responsible for verifying that the requesting user has passed the identity check, or must be re-directed to a login page. Moreover, it is design to accommodate different schemes of user authentication, for examples, id/password and digital certificate. This is easily achieved through aspect extension as shown by the following code skeleton for authentication aspects².

```
public abstract aspect Authentication {
    abstract pointcut pc(..); //arguments omitted
    abstract protected String forwardPage();
    abstract protected Boolean
        isAuthenticated(HttpServletRequest request);
    private ActionForward forwardTo
        (ActionMapping mapping,
         String signonPage) { //re-direct
        return mapping.findForward(signonPage);
    }
    ActionForward around(..):pc(..) //advice
    {
        if (isAuthenticated(request))
            return //continue
                proceed(mapping,form,request,response);
        else //login re-direction
            return forwardTo(mapping, forwardPage());
    } ...
}
public abstract aspect PWD extends Authentication
{
    protected String forwardPage()
    { return "globalSignOn"; }
    protected Boolean
        isAuthenticated(HttpServletRequest request)
    { ... } // if authenticated using password
}
public abstract aspect DC extends Authentication
{
    protected String forwardPage()
    { return "RequireDigitalCertificate"; }
    protected Boolean
        isAuthenticated(HttpServletRequest request)
    { ... } // if authenticated using certificate
}
```

Each sub-aspect of Authentication must define how to verify that a user has passed the proper identity check in the `isAuthenticated()` method via the session object, and where to re-direct the user if needed. Note that user re-direction is easily achieved through the action mapping mechanism of Struts. Finally, for each authentication scheme, there will be one concrete aspect that defines the pointcut designators specifying which actions to check.

² For space's sake, we omit the four arguments exposed to advice.

4.1.1.1 Precheck Aspects

We associate user actions that need authorization check with proper precheck aspects to ensure that the designated actions get executed only when a given constraint is satisfied. The following is the code for the *Precheck* aspect, the root of all precheck aspects.

```
public abstract aspect Precheck {
    abstract pointcut pc(..); //arguments omitted
    abstract protected Boolean
        constraint(HttpServletRequest request);
    abstract protected String getErrorMessage();
    private ActionForward forwardToErrorPage(..)
    { request.setAttribute
        ("message", getErrorMessage());
        return mapping.findForward("failure");
    }
    ActionForward around(..):pc(..)//advice
    {
        if (constraint(request))
            return //continue
                proceed(mapping,form,request,response);
        else //login redirection
            return forwardToErrorPage(request,mapping);
    } ...
}
```

The definitions of specific pointcuts and access constraints are left to concrete aspects that inherit it. For example, the following VIPOrder aspect enforces the VIP rule stated in Section 2.

```
public aspect VIPOrder extends Precheck {
    pointcut pc():
        execution(public ActionForward
            NewOrderAction.execute(..)
            && args(mapping,form,request,response);
        protected boolean constraint(request)
        { // get function arguments
            UserAccount userAcct =(UserAccount)
                request.getSession().getAttr("userAccount");
            CartForm cartForm = (CartForm)
                request.getSession().getAttr("cartForm");
            return (userAcct.getAttr("VIP")
                || cartForm.getCart().getTotal()<100000);
        }
    protected String getErrorMessage()
    {
        return "Only VIP customers can create orders
            whose total amount exceeds $10,000!!";
    }
}
```

It is quite clear that this enforcement aspect has a direct correspondence to the associated access control rule.

4.1.1.2 Postfilter Aspects

The postfilter aspect will filter out unauthorized data records from a user query result, and, if necessary, mask out sensitive data fields. As the code structure for handling a single record retrieved by key-based queries is different from that of handling a collection of data obtained by ad-hoc queries, we further divide the postfilter aspect into two styles: *single* and *collection*. The following is the code for the *PostfilterCollection* aspect³. It retrieves the query results from the request object and iteratively applies the filter and the field mask to each individual data record.

³ It can also be implemented as an after advice.

The definitions of specific filter condition and field mask are left to concrete aspects that inherit it.

```
public abstract aspect PostfilterCollection {
    abstract protected boolean filter
        (HttpServletRequest request, Object data);
    abstract protected Collection
        getRS(HttpServletRequest request);
    protected abstract void mask(Object data);
    private void remove(Iterator i) {i.remove();}
    ActionForward around(..) : pc(..)
    {
        ActionForward forward = proceed(..);
        Collection col = getRS(request);
        Iterator i = col.iterator();
        while(i.hasNext())
        {
            Object data=i.next();
            if (!filter(request, data))
                remove(i); // unauthorized records
            else mask(data); // fields masking
        }
        return forward;
    }
}
```

The code for `PostfilterSingle` is very similar yet simpler, since only one data record is being examined.

A distinguishing feature of the aspects defined above is that they are all stateless and singleton aspects. Unlike prior approach [4][5], we do not need to associate any user profile information with the authentication aspects. This is mainly due to the accepted convention in Web application development that user-related information is stored in the session object which is available in the aspect code through the exposed request object argument.

Finally we make some note about the advice type we chose. One may argue that the authentication and precheck aspects should be implemented as before advice instead of around advice. We disagree with this for two reasons. Firstly, by definition, in normal case before advice cannot prevent the intercepted method from execution, yet this is exactly what access control enforcement requires. Secondly, throwing access denied exceptions in before advice significantly increases the complexity of exception handling in the target application, and may make the aspects invasive.

4.2 Aspect Composition Issues

Having defined these abstract aspects, we can then apply them to implement the access control rules specified by security administrator. This can be done by defining concrete aspects inheriting from them that include the proper pointcut designators and the required constraint or filter methods. However, there are some conditional execution dependencies between these access control aspects. In particular, when required at the same join points, the `Postfilter` aspects should not be executed before the `Precheck` aspect finished successfully, whose execution in turn should be preceded by that of an `Authentication` aspect. These restrictions are very close to the $cond_{hard}(x, y)$ constraint defined in [14]⁴.

The straightforward implementation described above simply cannot ensure that these constraints are satisfied. We must use the “declare precedence” construct of AspectJ to enforce the ordered execution of these three aspects. Yet this may not completely

⁴ More will be said about the $cond(x, y)$ constraint in Section 5.

resolve the issue, for it is not uncommon that, during program maintenance time, an authentication aspect is carelessly left out or misplaced at a join point where certain access control is required. This would violate the conditional execution dependency. To avoid the damage which may result from such human errors, we decided to look for alternative implementations.

4.3 Alternative Implementation

One way to achieve our goal is to use per-object stateful authentication aspects to store user authentication records, as suggested by [14]. But this would make our code more complicated and lead to a proliferation of unnecessary aspect instances, as we stated above that we can take advantage of the session object used in Web applications for this purpose. Hence we decided to take another route to resolve the composition issue.

Ideally, we wish to be able to construct a composition aspect out of an authentication aspect, a precheck aspect, and maybe a postfilter aspect to ensure the conditional execution dependencies between them. Yet AspectJ has no such aspect composition mechanism for our purpose, so we choose to use aspect extension and other OO techniques to implement it. All the four abstract aspects defined earlier must be revised. In addition, we added a new abstract aspect, `AAAspect`, and a few helper classes to fulfill our requirements. They form a small aspect framework. Figure 3 illustrates the relationships among these aspects and classes.

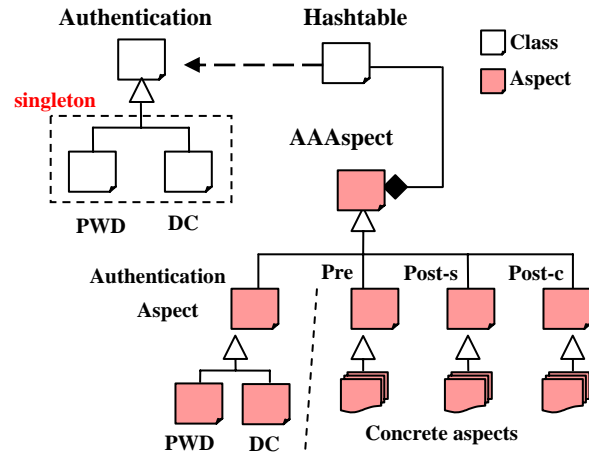


Figure 3: The Structure of the Alternative Aspects

The `AAAspect` is the root aspect that acts as a factory for producing the required authentication verification objects:

```
public abstract aspect AAAspect {
    abstract pointcut pc(..);
    abstract protected String getAuthType();
    private static Hashtable authTable
        = new Hashtable();

    static {
        authTable.put("PWDAuth", PWDAuth.getInstance());
        authTable.put("DCAuth", DCAuth.getInstance());
        ... // for other authentication schemes
    }
    protected Authentication getAuthObj() { //factory
        return (Authentication)
            authTable.get(getAuthType());
    }
}
```

The `getAuthObj()` method is the factory method that will return the right type of authentication verification object according to the result of executing the `getAuthType()` method, which is defined in sub-aspects.

The Authentication aspect extends the `AAAspect`. It factors out the common code structure of conducting user authentication verification check and leaves to its sub-aspects, e.g., `PWDAuthAspect`, the specifications regarding where to weave in the checking code (pointcut) and which type of authentication to use. For each authentication type, there will be an authentication object containing the code to verify that a user has passed the authentication check, e.g., `PWDAuth`. They are instantiated from subclasses of the `Authentication` class, which prescribes the interface for confirming authentication checks (i.e., `isAuthenticated()`). As the confirmation check is independent of application state, all authentication objects are instantiated once only and accessed through a static method, `getInstance()`, following the Singleton pattern [7]. Furthermore, each authentication object will specify its own user authentication page for login re-direction.

In this alternative implementation, the advice in all of the revised `Precheck` and `Postfilter` aspects must first get a proper authentication object and make a call to its `isAuthenticated()` method. This will ensure that access control enforcement is preceded by a proper user authentication. Similarly, the advice in the postfilter aspects must, in addition, include the code of access constraint check before proceeding to execute the intercepted method and filter out unauthorized contents. For space's sake, we show only the code for the revised `PostfilterCollection` aspect:

```
public abstract aspect PostfilterCollection {
    abstract protected Boolean
        constraint(HttpServletRequest request);
    abstract protected String getErrorMessage();
    abstract protected boolean filter
        (HttpServletRequest request, Object data);
    abstract protected Collection
        getRS(HttpServletRequest request);
    protected abstract void mask(Object data);
    private void remove(Iterator i) {i.remove();}
    ActionForward around(..) : pc(..)
    {
        Authentication auth = getAuthObj();
        if (auth.isAuthenticated(request)) {
            if (constraint(request)) {
                ActionForward forward = proceed(..);
                Collection col = getRS(request);
                Iterator i = col.iterator();
                while(i.hasNext()){
                    Object data=i.next();
                    if (!filter(request, data))
                        remove(i); // unauthorized records
                    else mask(data); // fields masking
                }
                return forward; // completed
            } else // access denied
                return forwardToErrorPage(request, mapping);
        } else // login re-direction
            return mapping.findForward(auth.forwardTo());
    }
}
```

The conditional execution dependencies are ensured by duplicating the code from the authentication and the precheck aspects.

5. DISCUSSION

Software Engineering Properties

As demonstrated above, the MVC architectural style lays a good foundation for modularizing access control. And we can take full advantage of the MVC-based Struts framework to achieve a complete separation of access control enforcement. Furthermore, the way an access control aspect is implemented have a direct correspondence to an access control rule. These features definitely make our access control implementation qualify for software engineering properties such as comprehensibility, analyzability, and adaptability.

Our aspects also support very good evolvability since the main dependency between our approach and Struts is the interface of user action (i.e. the `execute()` method). As long as it remains unchanged, it is very unlikely that we need to change our approach. For example, since Version 1.1, Struts also supports method-based dispatch units through a new class called `DispatchAction`. Yet, all those dispatchable methods in a dispatch action class must have the same signature as the `execute()` method. Hence all we need to change is the method names in pointcut designators.

Our aspect framework supports multiple user authentication types. Not only function-level access control but also data filtering can be easily realized in a non-invasive manner. We argue that we have achieved a very good balance between serving the needs of a wide range of access control requirements and that of incurring minimal impact on the structure of underlying applications. As a case study, we have successfully applied this aspect framework to re-engineer the access control part of an open-source B2C E-Commerce application, `JPetstore` [12] with very little adaptation.

Aspect Composition

Nevertheless, in the process we did feel confused by issues of aspect reuse and aspect composition. As discussed earlier, at shared join points, we must have all the required aspects and maintain the conditional execution dependencies among them. Yet, in our first implementation, the authentication, precheck and postfilter aspects are defined separately and get applied to each join point individually. This may lead to undesirable runtime exceptions or even security breach. For lack of proper aspect reuse and composition mechanisms in `AspectJ`, we are forced to design an alternative implementation where the code that should be reused (i.e., authentication and precheck aspects) is simply duplicated in the composition aspects (i.e., precheck and postfilter aspects.)

Earlier work on aspect reuse in `AspectJ` [10][19] did not properly address the issues involved. The recent work of Nagy et al [14] introduced the notion of control constraint, `cond(x, y)`, between two aspects x and y , which to a very large extent clarified the essence of this issue. However, we still do not feel the issue is completely resolved. In particular, the definition of `cond(x, y)` focuses on only the dependencies between two aspects at shared join points; nothing about their relations with the intercepted method is considered. This fits well with the behaviors of before

and after advice, but not with that of *around advice*. By nature, around advice concerns the execution of the intercepted method and it is the internal logic of around advice that will determine whether to resume the intercepted method, or continue executing the next advice in the case there are multiple around advice anchored at same join point. Therefore, we claim that the *cond(x, y)* constraint does not completely capture the conditional execution dependency between two around advice.

The peculiarity of around advice leads us to re-think what the proper forms for aspect composition should be and what linguistic mechanisms an AOP tool should provide for them. Recently, some new Java-based AOP tools, instead of proposing language extensions, take the approach of supporting AOP via runtime libraries and, in some sense, blur the distinction between aspects and classes. Just before sending out this position paper, we ran across a radical proposal that advocates the unification of aspects with classes [15]. Does this mean that we could compose aspects just like how we do for objects?

JBoss AOP

We decided to take a quick look into these tools and chose the recently released JBoss AOP [11] as our target of investigation. We report our preliminary findings as follows. There is no new aspect abstraction in JBoss AOP; any Java class can become an aspect as long as it implements a special interface, namely `Interceptor`⁵, and puts the advice code in a special method called `invoke`. The pointcut designators are separated from aspect definition and defined in an XML configuration file, in which the target classes and the interceptors to weave in are linked. The advice in an aspect (interceptor) all takes the form similar to the *around* advice in AspectJ. For example, the following is the code skeleton for authentication verification in JBoss AOP.

```
public abstract class Authentication
    implements Interceptor {
    abstract protected String forwardTo();
    abstract protected boolean
        isAuthenticated(HttpServletRequest request);
    // advice body
    public Object invoke(Invocation invocation)
        throws Throwable {
        Object[] arguments = ((MethodInvocation)
            invocation).getArguments();
        ... // get info exposed at join point
        if (isAuthenticated(request))
            return invocation.invokeNext();
        else
            return // login re-direction
                forwardToSignonPage(mapping, forwardTo());
    }
    ...//some auxiliary definitions
}
```

The invocation object passed to an advice is similar to the `thisJoinPoint` object available in an advice of AspectJ. In addition, it has an `invokeNext()` method that does the same thing as the `proceed()` does for around advice in AspectJ.

⁵ We used an earlier version of JBoss AOP for our study. The latest version (1.0.0 Final) has made some generalizations: an aspect can include multiple advices, yet all of which must have a specific signature, and an interceptor is an aspect with only one advice named "invoke".

Basically, both aspect suites defined earlier can be easily adapted to JBoss AOP. Yet our goal is to see whether those access control aspects can be composed in an object-oriented manner, since aspect instances in JBoss AOP are just like object instances. However, it turns out that the nature of around advice makes this attempt fruitless. Firstly, we tried inheritance and defined the precheck aspect as a sub-aspect of the authentication aspect. In order to reuse the authentication advice in the precheck aspect, we have to twist the code of both aspects⁶. Secondly, we tried the HAS-A composition and embedded an authentication aspect instance into the precheck aspect. Again, we have to make some workarounds that change both aspects to compose them properly⁷.

Despite these unsuccessfully efforts, we continued exploring the undocumented API of JBoss AOP. What we find interesting is that there is another `invokeNext(Interceptors [])` method declared in the `Invocation` interface of JBoss AOP. It enables us to define composite aspects and install them at selected join points just like atomic aspects. For instance, the following is the code for a generic composite aspect:

```
public abstract class Composite
    implements Interceptor {
    //constituent aspects
    protected Interceptor[] intcptArray;
    // assemble other aspects into "intcprArray"
    abstract protected void assembleAdvice();
    public String getName() {
        return "CompositeAspects";
    }
    public Object invoke(Invocation invocation)
        throws Throwable {
        assembleAdvice();
        return invocation.invokeNext(intcptArray);
    }
}
```

It simply assembles a composite aspect out of an array of interceptors and uses the `invokeNext(Interceptors [])` method to trigger the execution of the constituent advice⁸. The VIPOrder rule stated earlier can then be implemented by composing an authentication aspect and a precheck aspect in a straightforward manner as follows:

```
public class VIPCombo extends Composite {
    public String getName() { return "VIPCombo"; }
    protected void assembleAdvice() {
        PWDAuth auth = PWDAuth.getInstance();
        VIPOrder vip = VIPOrder.getInstance();
        intcptArray = new Interceptor[] { auth, vip };
    }
}
```

⁶ Afterwards, we realized that what we did is something like emulating the "inner" construct of Beta [13] in Java. Inheritance in Beta assumes a different semantics. An inherited method in a superclass has precedence over the inheriting method in a subclass, and may refer the inheriting one through the "inner" construct [3]. This discovery seems worth further study.

⁷ We overload the authentication advice with an extra parameter and pass the proper "this" argument to it so that we can choose the right proceeding path to follow in the advice. This is similar to the situation described in [6], where object composition and message forwarding are used to simulate inheritance.

⁸ This is reminiscent of the *chained advice* described in [9], though no concrete implementation in AspectJ is provided therein.

Nevertheless, we encountered two issues that make us think that this feature of JBoss AOP may not be designed with the intention we had wished for. The first one is about aspect instantiation which is not so essential to the composition issue and thus omitted here. The second issue matters most since we found that the mechanism we used above is not truly compositional. In particular, we cannot compose an aspect out of atomic aspects and composite aspects arbitrarily. In a composite aspect, only the last constituent aspect can be composite, too. These findings confirm our thinking that the issues of aspect composition are far from trivial and merit further investigation.

6. CONCLUSION

In this position paper, we have reported our experience in designing access control aspects for Web applications. We showed that the MVC architectural style lays a good foundation for modularizing access control, and we can take full advantage of the MVC-based Struts framework to achieve a complete separation of access control enforcement. The access control aspects we presented have achieved a very good balance between serving the needs of a wide range of access control requirements and that of incurring minimal impact on the structure of underlying applications.

On the other hand, we have also identified the issues of reusing and composing around advice. We described the problems we ran into while designing the access control aspects. An alternative aspect framework was developed that achieved guaranteed conditional execution at the cost of code duplication. Along the line we further explored the issues in the context of JBoss AOP, where aspects are plain Java classes with methods of a specific signature. We found that conventional object composition and inheritance cannot support advice reuse directly as we would expect. In contrast, Beta-like inheritance and its inner construct may suit our purpose. Furthermore, we pointed out the asymmetric nature of the chained aspect composition mechanism provided by JBoss AOP. Although these preliminary findings are not very positive, yet they have shed some light on the strong and weak points of the aspect composition mechanisms provided in current AOP languages and tools.

7. REFERENCES

- [1] The Apache Struts Web Application Framework: <http://struts.apache.org/>
- [2] The AspectJ website: <http://www.eclipse.org/aspectj/>
- [3] Bracha, G. and Cook, W., Mixin-based inheritance. In *Proc. ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, October 1990.
- [4] De Win, B., Joosen, W., and Piessens, F., AOSD & Security: a practical assessment, *Workshop on Software engineering Properties of Languages for Aspect Technologies (SPLAT03)*, 2003, pp. 1-6.
- [5] De Win, B., Vanhaute, B., and De Decker, B., How aspect-oriented programming can help to build secure software, *Informatica* vol.26(2), 2002, pp. 141-149.
- [6] Fröhlich, P.H., Inheritance Decomposed, *Inheritance Workshop, European Conference on Object-Oriented Programming (ECOOP)*, Malaga, Spain, 11 June 2002.
- [7] Gamma, Helm, Johnson and Vlissides: *Design Patterns*. A. W. L., 1995. ISBN 0-201-63361-2.
- [8] Goodwin, R., Goh, S.F., and Wu, F.Y. Instance-level access control for business-to-business electronic commerce, *IBM System Journal*, vol. 41, no. 2, 2002.
- [9] Hanenberg, S. and Schmidmeier, A., Idioms for Building Software Frameworks in AspectJ, *2nd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, Boston, MA, March 17, 2003.
- [10] Hanenberg, S. and Unland, R., Using and Reusing Aspects in AspectJ, *Workshop on Advanced Separation of Concerns in Object-Oriented Systems, OOPSLA*, Oct. 2001.
- [11] JBoss AOP website: <http://www.jboss.org/products/aop>, document download site: <http://docs.jboss.org/aop/aspect-framework/>
- [12] Johnson, R. *The JPetstore Demo Application*, Shipped with the Spring Framework. <http://www.springframework.org/>
- [13] Madsen, O.L., Moller-Pedersen, B., and Nygaard, K., *Object-Oriented Programming in the Beta Programming Language*, Addison-Wesley (ACM Press), 1993.
- [14] Nagy, I., Bergmans, L., Aksit, M., Declarative Aspect Composition, *Workshop on Software engineering Properties of Languages for Aspect Technologies (SPLAT04)*, 2004.
- [15] Rajan, H. and Sullivan, K. Classpects: Unifying Aspect- and Object-Oriented Language Design, *Technical Report CS-2004-21*, Department of Computer Science, University of Virginia, Sept 2004.
- [16] Sun Microsystems, Java Authentication and Authorization Service (JAAS), <http://java.sun.com/products/jaas/index.jsp>
- [17] Vanhaute, B., De Win, B., and De Decker, B., Building Frameworks in AspectJ, ECOOP 2001, *Workshop on Advanced Separation of Concerns*, pp.1-6.
- [18] Verhanneman, T., Jaco, L., De Win, B., Piessens, F., and Joosen, W., Adaptable Access Control Policies for Medical Information Systems, *Proc. of Distributed Applications and Interoperable Systems*, 2003, Paris, France, LNCS 2893, pp. 133-140.
- [19] Wohlstadtter, E., Keen, A.W., S. Jackson, and Devanbu, P., Accommodating Evolution in AspectJ, *Workshop on Advanced Separation of Concerns in Object-Oriented Systems, OOPSLA 2001*, October 2001