Modularizing Security Related Concerns in Enterprise Applications – A Case Study with J2EE and AspectJ

Renat Zubairov, Stefan Hanenberg, Rainer Unland
University of Duisburg-Essen,
Institute for Computer Science and Business Information Systems,
Schützenbahn 70, D-45117 Essen, Germany
{shabenbe, unlandr}@cs.uni-essen.de
renat.zubairov@uni-duisburg.de

Abstract: Security concerns are one of the predominant concerns for the development of web applications – applications that are written in a certain architectural context like J2EE. However, the crosscutting nature of security implementations in such enterprise frameworks is hardly studied. Furthermore, the impact of using aspect-oriented techniques in order to implement security relevant concerns in a J2EE architecture is hardly studied. Consequently, developers cannot decide upfront whether or not the application of aspect-oriented techniques for implementing security concerns has a significant positive impact on the resulting application. This paper studies the impact of implementing security relevant concerns in a J2EE context via a case study. Security concerns for a given base application are implemented in a pure J2EE style as well as in an aspect-oriented style using AspectJ. Both approaches are compared and assessed in respect to the necessary changes in the base application.

1 Introduction

Security considerations are one of the predominant concerns for the development of for example web applications – data confidentiality is necessary in order to transmit passwords, authentication is required to provide user-specific data, authorization determines whether a certain user is allowed to access certain information. The underlying architecture for such application is typically an enterprise framework such as J2EE [14] for the programming language Java.

For security related issues there exist already a number of frameworks which encapsulate the corresponding domain knowledge (like JAAS [15], JCA [16], etc.) and which need to be integrated into the enterprise environment being used. Although such frameworks already encapsulate security-specific domain knowledge, it does not automatically imply that the corresponding concern implementations within the resulting applications are well modularized. They potentially cause tangled code: The API calls to such frameworks are somehow mingled with the business logic.
Aspect-oriented software development [2, 8] addresses the problem of crosscutting concerns, i.e. concerns whose implementations inherently cannot be modularized using given base language constructs. In order to modularize such concerns, aspect-oriented systems provide additional composition mechanisms namely aspects which modularize crosscutting concerns by providing means to adapt join points –which are “principled points in the execution of a program” [9].

However, developers need to learn a number of new constructs and abstractions, which is a time (and money) consuming task. For example in AspectJ (cf. e.g. [11]) developers need to learn the language constructs pointcut, advice and introduction.

In order to determine whether it is reasonable to handle security related implementations using aspect-oriented techniques developers have to analyze whether the security related concerns really crosscut the application. Next, they have to determine whether the crosscutting is acceptable in a way that the code is still readable and maintainable. If this is not the case the developers have to estimate whether aspect-oriented techniques like AspectJ help to handle the crosscutting problem in a significant way.

While some concerns like for example persistence (cf. e.g. [13]) are already exhaustively discussed, the crosscutting characteristics of security concerns in the architectural context of J2EE are rather not known in detail. Hence studies are needed that analyze the impact of security concerns in J2EE using conventional composition mechanisms as well as using aspect-oriented techniques.

This paper presents a case study for security related implementations for a given J2EE application. The intention is to keep the application quite simple but still relying on typical J2EE configurations. We describe a number of typical security related requirements for the underlying application and implement those using conventional J2EE techniques as well as using AspectJ as a representative for an aspect-oriented system. Then we discuss both implementations in respect to their applicability.

Section 2 introduces a blog application and discusses corresponding security requirements. Section 3 illustrates the implementation of security requirements in a pure J2EE context. Afterwards, section 4 discusses the corresponding AspectJ-based implementation. In section 5 pros and cons of both implementations are discussed. After referring to some related work in section 6, the paper is finally concluded.

2 Example application

The example application is a blog management application. A blog is basically a journal that is available on the web. The activity of creating and maintaining the blog is blogging and someone who keeps a blog is a blogger. Blogs are typically daily processed using software that allows people with little or no technical background to update and maintain blogs. Postings on a blog are arranged in a chronological order with the most recent addition first.
2.1 Core functionality and architecture

Figure 1 illustrates the use cases of the system. Such use cases are common for most of the multi-user blog applications like Blogger\(^1\) or LiveJournal\(^2\). The actors are: Anonymous reader (every non-authenticated user having a web-access is a potential anonymous reader), Authenticated reader (a user who is registered in the system and authenticated), and Blogger (who is a registered and authenticated user who is allowed to create and maintain its own blogs). The use cases describe the way in which the actors interact with the system:

- All actors may read posted blogs without any restrictions.
- Authenticated users are furthermore permitted to post comments or to become a blogger.
- Comments made on a blog’s entry can be deleted by the authenticated user who created it or by the entry’s author.
- Every blogger is able to create a new blog.
- Every blogger can create or delete entries in his blog and delete comments that are left by other users.

The application consists of a number of non-functional requirements like remote access via a RMI client (clients that access the blog via an applet or an ordinary distributed Java application) or via a web interface. Furthermore, there are persistence requirements because blogs are stored in the external storage. Since such requirements are typical prerequisites for the application of enterprise frameworks their usage seems to be suitable. The application is to be written in Java. Therefore it is reasonable to use the J2EE framework [14].

The domain model is built up in a straightforward way. It mainly consists of three classes, Blog, Entry and Comment. Blogs contain entries on which comments may be added. The representation and the model are connected based on the Model-View-Controller pattern (MVC, [1]) – model classes are modified by controllers (shared between GUI and web clients) and rendered by different views (like HTML elements or web pages or UI element). In the core application functionality no security related implementations exist – there are no authentication windows, no distinctions between different actors, etc. Every user can execute all actions.

The blog application is implemented using J2SE and J2EE technologies and contains the following tiers:

- The client-tier is either a Swing-based graphical user interface or an HTML browser that renders HTML pages.

\(^1\) http://www.blogger.com
\(^2\) http://www.livejournal.com
The web-tier runs on a J2EE server. This tier is responsible for creating HTML pages that provides means for interacting with the system.

The business-tier runs on a J2EE server. This tier contains the business logic components, which encapsulate the main functionality.

The enterprise application integration tier runs on a different server containing the database.

The primary design pattern of the web-tier is the MVC. For implementing corresponding controllers Servlets [2] are used that are responsible for parsing HTTP requests, forward data to the application tier and pass the results to the view components. View components are implemented using the template engine Apache Velocity [17]. View components are responsible for data binding and generating HTML. For reasons of simplicity, the web tier is running on the same JVM as the core application. Therefore no special communication mechanism is needed. The GUI client is based on Swing and communicates with the server via RMI.

<table>
<thead>
<tr>
<th>Category</th>
<th>model</th>
<th>servlet</th>
<th>GUI</th>
<th>facade</th>
<th>velocity</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>57</td>
<td>98</td>
<td>496</td>
<td>16</td>
<td>79</td>
</tr>
</tbody>
</table>

The business logic layer is implemented using plain Java. The primary design pattern for this layer is the facade [3]. This layer operates on model objects and maps them to the database using the object-relational mapping framework Hibernate3.

3 http://www.hibernate.org
Conceptually the elements of the core application belong to one of the following five categories:

- **Model** contains all classes representing the domain model implemented in Java.
- **Facade** provides the business logic that is used from web clients (made via servlets) and from GUI clients (made via RMI). Classes in this category are also responsible for calling the ORM tool.
- **Servlet** contains all servlets as well as additional helper classes in order to receive and reply requests of web clients. This category contains only code for data binding and makes use of the facade classes.
- **GUI** contains all classes that are related to the GUI client implementation.
- **Velocity** contains all velocity templates used for the HTML representation.

Figure 22 summarizes the core application without any implemented security concerns in terms of lines of code. According to the quite simple domain model, the model classes are just 57 lines of code. The facade category just consist of 16 lines of code, the servlet implementation requires 98 lines of code. The GUI layer consists of 496 lines of code and the velocity category has 79 lines of code.

### 2.3 Security requirements

In addition to the security requirements that are implicitly defined in the use cases the following requirements need to be considered:

1. **Explicit security policies for authentication and authorization**: All actions done by the user should be executed according to explicitly defined policies.
2. **Lazy authentication**: Authentication is done only when needed, for example authentication is not needed for reading the blogs. Furthermore, the client application is acting on behalf of one user at a time.
3. **Instance level authorization**: Authorization is done based on a per object basis but not on a per-class basis. For example, only a blog owner may post entries in his own blog, while comments may be deleted by their owner (creator).
4. **Non-Repudiation**: The sender is provided with a proof of delivery and the recipient is provided with a proof of the sender's identity so that no one can later on deny having processed the data.
5. **Data confidentiality protection**: The channel between the client and the server should be secure and not being interpretable by a third party.
6. **UI filtering**: The user interface should be changed according to the current user’s authorization level. For example, if a user is not permitted to delete a blog then the “Delete blog” button/menu items should be hidden or disabled.

---

*We calculated the whole number of lines in the template files.*
7. Auditing: Application objects should be logged in terms of who accessed them and when they were accessed.

The J2EE platform includes already technologies for a large number of middleware services required for enterprise-level applications in order to implement security concerns. The Java Authentication and Authorization Service (JAAS, [15]) is a basic technique for authentication and authorization. The Java Cryptography Architecture (JCA, [16]) defines a generic framework for accessing and developing cryptographic functionality for the Java platform.

3 Pure J2EE solution

This section discusses implementation details of the pure J2EE solution for the security relevant code. First, we describe how authentication was implemented. Then, we describe how we implemented the authorization concern followed by the data confidentiality protection. Finally, we discuss the implementation of auditing.

Authentication

For the implementation of the authentication based on JAAS we needed to consider two tiers involved – the client-tier and the web-tier.

```java
protected void doGet
   (HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException {
   .. Parsing parameters from request(newBlock, deleteBlock)
   // reads the current user's role from the session
   Subject sub = request.getSession().getAttribute(SUBJECT_KEY);
   try {
      if (newBlog != null) { ... Creating new blog ...}
      if (deleteBlog != null) { ... Deleting blog ...}
   } catch (bundle) ... exception handling if no subject found
}
```

Fig. 3. Exemplary subject request in servlet.

For the GUI part, authentication windows and error message dialog boxes were introduced. Authentication method calls were introduced before every action that requires authentication. The authentication context is managed on a per-JVM basis (one GUI client instance acts on behalf of one user at a time) therefore the context is stored using the Singleton design pattern [1].

182
For the web tier login and error pages were created by creating corresponding velocity files. The authentication on the web tier cannot be handled via a singleton since multiple clients access the same servlet instance. Therefore an authentication context is stored in the HTTP session. Within the servlets special method calls were introduced before any authentication sensitive action. Such methods retrieve the user identity from the current session. If no such identity is found an exception is thrown. Since one servlet is responsible for all actions associates with a certain model class (like Blog) it contains multiple places where authentication calls were introduced.

Figure 3 illustrates a code fragment illustrating the adaptation of the web tier in order to implement the authentication concern: Before executing any action the current subject (which is the user’s authentication according to JAAS) needs to be requested and additional exceptions need to be handled in case such a subject is not available.

2 Authorization

In order to implement instance-based authorization several additional associations in the model classes were required because of the required ownership information. The default implementation of JAAS is not suited for an instance-based authorization, but its open architecture allows an appropriate seamless integration. Authorization calls needed to be added before any authorization sensitive operations. Authorization checks were added in all controllers (according to the MVC-architecture). All authorization checks contain two parts – creation of a permission object that encapsulates the needed permission, and delivery of the permission object to JAAS.

```
1 ...
2 Subject subject = ...
3 Permission permission = new DeletePermission(thisBlog);
4 .. perform an authorization check via JAAS...
5 .. perform an action
6 ...
```

Fig. 4. Exemplary permission check for a delete operation on a given blog.

Figure 4 illustrates the code for requesting permissions for a delete operation on a given object (represented by thisBlog). The same checks were added to all places where specific permissions were needed. The authentication context needs to be associated with the execution context. The JAAS approach relies the command pattern in conjunction with a strategy pattern [1] – the command implements the action to be performed, the strategy implements the permission lookup, the subject lookup and the comparison of both.

In case the current user does not have appropriate rights to perform the requested action, additional runtime exceptions are thrown. This also requires additional exception handling to be performed in order to show meaningful error messages.
Another facet of authorization is UI filtering. The user interface should not include buttons or links that lead to authorization errors (if a user cannot delete a blog, then the “Delete” button should not be shown). For non-web clients this feature was implemented using listeners that check the permission for the currently selected object and update the graphical user interface in an appropriate way. For the web tier authorization checks needed to be added to the Velocity templates.

3.4 Data confidentiality protection and non-repudiation

Based on the system architecture there are two communication channels that need to be protected – the RMI communication channel between the server and the GUI client and the HTTP channel between the web client and the web tier. Solutions for data confidentiality protection and non-repudiation are well known and based on SSL: HTTPS for HTTP and RMI over SSL for RMI.

Using HTTPS requires no changes in the web tier at all because all changes are done in the servlet container configuration. Additional security enforcements for the RMI communication channel also require almost no changes in the code - only one additional initialization parameter needs to be set. All related features like encryption, non-repudiation, key-store management etc. are managed transparently by JCA – correspondingly almost no change of the application code was necessary.

3.5 Audit

For implementing audit we extended the Log4J [10] framework. First, additional classes representing the audit logging level and the audit event were created. Also, some special classes for gathering audit information (audit appender, see [10]) were created. Examples of audit information are method names and class names.

```java
0 public void createBlog(...) {
1  if (log.isEnabledFor(AuditEvent.AUDIT)) {
2    AuditEvent e = new AuditEvent(log, "Creating a new blog");
3    e.setOperationName("createBlog");
4    e.setAffectedObjectId(newBlog.getId());
5    e.setClassName(getClass().getName());
6    log.callAppenders(e);
7  }...
9 }
```

Fig. 5. Audit code for blog creation.

For adding the audit to the application, all methods that need to be audited required some extra code – the creation and initialization of the new audit event and the invocation of the corresponding audit appender.

For the web tier not all information (like for example authentication information) is available in the place where it is needed. Therefore the mapped diagnostic context (MDC) was used. MDC is a threaded storage for arbitrary data. This data is added to
every logging statement happening in the current thread. We used MDC for storing authentication context therefore all subsequence logging calls that occur in the current thread were logged with corresponding authentication information.

### 3.6 Summary

The J2EE solution fulfills the security requirements. The enterprise APIs like JAAS, JCA and Log4J were applied to implement the security related concerns. Consequently, we did not need to analyze and design a number of elements on our own. For example, we did not need to design the representation of users (which corresponds to the type `Subject` in JAAS), or the representation of permission objects (which corresponds to the type `Permission` in JAAS). Also, a large number of auditing related tasks was already performed by Log4J — the representation of the logger, the design of `AuditEvents` (although they needed to be implemented on our own) opening and closing the event storage, etc. From that point of view the application of the APIs can be considered to be successful since they provided an appropriate representation of security related domain knowledge.

In order to compose the security concerns (or the different facets of the security concern) with our base application we needed to perform a huge number of changes in our application. For example, we needed to refactor a large number of elements in the web tier: The actions like adding or removing a blog that were originally directly implemented in corresponding classes (like `Blog` in this example) needed to be shifted into different modules due to the JAAS design decision to apply the command and strategy design patterns.

<table>
<thead>
<tr>
<th>Package</th>
<th>model</th>
<th>facade</th>
<th>servlets</th>
<th>GUI</th>
<th>velocity</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>61</td>
<td>58</td>
<td>196</td>
<td>558</td>
<td>108</td>
</tr>
</tbody>
</table>

Fig. 6. Application categories in J2EE security implementation

The overall impact of the J2EE implementation on the application is illustrated in Figure 6 in terms of lines of code. In addition to the growth of the application because of adding security relevant concerns, the code turns out to be highly crosscutting – almost none of the original modules could be used as is and destructive modifications needed to be performed in almost all of them.

### Aspect-oriented solution

This section describes the aspect-oriented solution of the same security related requirements based on AspectJ. In correspondence to the previous section this section is structured according to the different facets of the security concern – authentication, authorization, data confidentiality protection, and auditing.
4.1 Authentication

In order to modularize the user authentication for the two different client environments a common abstract aspect was created. The aspect’s intention is to encapsulate common features like an abstract pointcut demarking those join points where authentication information should be available. The authentication context is propagated on the server side via an application of the \textit{wormhole} pattern [11]: The authentication context is being passed along the control flows via pointcuts that use the \texttt{cfow} pointcut designator. The RMI client still explicitly needs to pass the user authentication information to the server via an additional parameter.

The extraction of the subject from a HTTP-request was implemented via a corresponding advice whose join point is the execution of the corresponding servlet. The code within the advice corresponds to the one illustrated for the J2EE solution in Figure 3 – it retrieves the subject from a session’s attributes.

4.2 Authorization

The aspect-oriented approach uses a different technique for implementing the ownership association. AspectJ permits to extent a class with additional members using introductions. Consequently, ownership relationships were introduced to the domain model classes using the \textit{container introduction} pattern [5]. Also the ORM mapping and the database schema were changed in order to store ownership relation in the DB, which corresponds to the necessary change in the pure J2EE solution.

\begin{verbatim}
1 pointcut criticalAccess(Object criticalObject, Subject subject):
2     criticalJoinPoint (criticalObject)
3     && AuthenticationAspect.authenticationCFlow(subject)
4     && !cfow(ORMOperations());
\end{verbatim}

Fig. 7. Exemplary pointcut for selecting domain object accesses.

The authorization related code in the aspect-oriented solution contains one aspect that encapsulates the general functionality of checking the authorization and two concrete aspects that encapsulate client specific authorization details. The abstract aspect defines a set of pointcuts for each permission type. Furthermore, it contains a set of before advices checking the authorization. All authorization related pointcuts are composed from three parts:

1. \textbf{Execution pointcut}: this pointcut identifies all join points where the domain model information is accessed.

2. \textbf{Control flow pointcut}: This pointcut is reused from the corresponding concrete authentication aspect described in the previous subsection in order to determine the current user accessing the domain model.

3. \textbf{Negated Control flow pointcut}: This pointcut definition prevents recursive calls via excluding all join points invoked within the control flow of ORM initiated actions.

186
Figure 7 illustrates an exemplary pointcut appearing in the aspect that handles authorization checks. The pointcut `criticalAccess` selects all method execution join points of critical methods (specified in the pointcut `criticalMethods`). The parameter `criticalObject` of the corresponding pointcut represents the domain object on which the critical operation is performed. The pointcut `authenticationCFlow` binds the subject to the user identity. The last pointcut excludes all join points occurring in the control flow of OR-mapping operations.

This pointcut is already sufficient to provide the context for computing whether a certain action is permitted. The only missing information is the permission object that can be computed from within the corresponding advice on the critical object – which corresponds to the code in the pure J2EE solution.

The implementation of the UI filtering functionality differs hardly in comparison to the pure J2EE solution. This is because on the web tier all changes are done within the Velocity files. On the GUI tier specially created selection listeners are advised by the authorization aspect.

4.3 Data confidentiality protection and Non-repudiation features

Data confidentiality protection and non-repudiation features were relatively well modularized using the pure J2EE solution. Only one additional initialization parameter needed to be set. Without discussing any details here we changed one line in the aspect-oriented solution (without using AspectJ language constructs).

4.5 Audit

Audit and logging in general is a prominent example used in the aspect-oriented literatures (see for example [7]). Hence, it is no wonder that the AspectJ implementation for connecting Log4J to our application significantly differs from the pure J2EE solution. We implemented one abstract auditing aspects that is being woven across different tiers.

The context information like the current user performing a certain action, etc. is extracted via an explicit parameter passed by a pointcut (which corresponds to a similar pointcut definition as explained in section 3.2). The current executing method is being computed from the special keywords `thisJoinPoint` or `thisStaticJoinPoints` that permit to reflect on the current join point. The code within the corresponding advice corresponds to the code being inserted in sections 3.5. Another task of the aspect is to maintain the reference to the logger. One characteristic of the auditing aspect is that it is completely orthogonal from the underlying application, i.e. no changes where necessary in the application code in order to provide auditing.
4.6 Summary

By using the aspect-oriented approach we were able to handle a large number of security relevant implementations using aspect-oriented constructs. Figure 8 concludes the lines of code needed to implement them on top of the blog application.

<table>
<thead>
<tr>
<th>Package</th>
<th>model</th>
<th>facade</th>
<th>servlets</th>
<th>GUI</th>
<th>velocity</th>
<th>aspects</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>57</td>
<td>20</td>
<td>98</td>
<td>512</td>
<td>108</td>
<td>29</td>
</tr>
</tbody>
</table>

Fig. 8. Application categories in AO security implementation

However, there are still some situations where simply using an aspect-oriented language did not help: For UI filtering in the web tier it is necessary to adapt some velocity files. Since velocity templates are not written in Java they cannot be handled using AspectJ. Although it is technically possible to adapt the behavior of velocity itself (instead of the application specific velocity files) in order to change the behavior of evaluation certain forms in certain contexts, this was not considered as a valid application of AspectJ. Also some additional changes in the OR mapping files needed: We had to add ownership relation there and update the database schema accordingly.

An interesting feature of the AspectJ solution is that the context information about the authorization and authentication was passed using the cflow constructs – there were almost no changes in any signatures necessary. The only exceptions are the RMI interfaces – user information needed to be explicitly delivered via special parameters. The aspect specification is rather small and consists of 29 lines of code for all aspects.

5 Discussion

Three different versions of the application have been introduced in the previous sections. First, the blog application without any security concerns. Second, the blog application including the security concerns written purely in a J2EE environment, and third, the same application including security concerns written in a J2EE environment with the aid of AspectJ. Figure 9 summarizes all three applications in terms of lines of code according to the categories GUI, servlets, facade, model and velocity. Furthermore the category aspects is added since the AspectJ solution provides a different conceptual model, the security concern, that is being implemented as a collection of aspects.

The lines of code in the categories facade and servlets differ widely between the different versions. The facade category represents the code for providing a façade corresponding to the façade design pattern whose main contribution is to provide a high level interface to the domain objects for the different tiers. The auditing as well as the authorization requirements in the pure J2EE solution forced to modify all existing methods in the façade and the resulting number of lines of code in the pure J2EE solution is almost 2.5 time higher than in the original application. The reason for this lies in the characteristics of a façade: The façade mainly forwards messages to different
objects and contains almost no functional code. Consequently, the code being added for the sake of auditing and authorization is much larger than the original code. In the servlets category the main problem in the pure J2EE solution was the context extraction and forwarding. Authorization information needed to be extracted and passed as an explicit parameter. Because of that, the security concerns in a pure J2EE solution can be considered to be highly crosscutting and the resulting code seems to be rather unacceptable. The changes in the GUI and the velocity category seem to be tolerable (in comparison to the categories servlets and façade).

<table>
<thead>
<tr>
<th>category</th>
<th>Blog without security concern</th>
<th>OO-Security</th>
<th>AO-Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUI</td>
<td>496</td>
<td>558</td>
<td>512</td>
</tr>
<tr>
<td>servlets</td>
<td>98</td>
<td>196</td>
<td>98</td>
</tr>
<tr>
<td>façade</td>
<td>17</td>
<td>58</td>
<td>20</td>
</tr>
<tr>
<td>model</td>
<td>57</td>
<td>61</td>
<td>57</td>
</tr>
<tr>
<td>velocity</td>
<td>79</td>
<td>108</td>
<td>108</td>
</tr>
<tr>
<td>aspects</td>
<td>N/A</td>
<td>N/A</td>
<td>29</td>
</tr>
<tr>
<td>Σ</td>
<td>747</td>
<td>981</td>
<td>824</td>
</tr>
</tbody>
</table>

Fig. 9. Summary of LOC of the base application

The aspect-oriented approach on the other hand turned out to be almost completely orthogonal to the categories façade, servlets and model. Especially the number of the lines of code in the servlets category is the same as in the original application. Within the GUI category also additional destructive changes were performed and an additional aspect was created to contribute to the events handling. Consequently, the solution is not clearly modular from the aspect-oriented point of view, but it still dominates the pure J2EE solution. Furthermore, the AspectJ solution introduces one new category (containing all aspects representing security related implementations), which is just 26 lines of code, and consists of 7 aspects.

All in one, the pure J2EE solution with security concerns causes an overhead of about 30% (981 LOC / 747 LOC) while the aspect-oriented approach causes only 10% overhead (in terms of LOC). Furthermore, the aspect-oriented approach allowed to keep almost all business logic on the server side as is – the design of the underlying application does not have to be learned and understood. In contrast to this, the pure J2EE solution introduced a large number of new design elements (commands, strategies, etc.), which makes it hard to understand the resulting application in terms of the domain model. In fact, all business logic becomes tangled with the security implementations.
Consequently, it is not possible to change or evolve the business logic without changing the underlying concern implementations.

According to the previous discussion we can conclude that the different facets of the security concern tend to be highly crosscutting even in the J2EE context. This observation is interesting, because applied APIs already encapsulate a large amount of the domain knowledge needed. However, the API calls itself are the crosscutting elements in the pure J2EE solution. Next, we can conclude, that the AspectJ-based solution essentially predominates the pure J2EE solution in terms of lines of code as well as in terms of reusability of the original application. Only the categories facade and velocity required some destructive modifications.

However, there is still a number of issues that were not well modularized from the aspect-oriented perspective. Although the authentication context propagation was nicely encapsulated in pointcut definitions on the server-side, the client-sided handling of the context propagation worked differently and required RMI interface adaptations. Probably, techniques like remote pointcut (as proposed in [12]) might allow a unique aspect definition for both the client side as well as the server side. Nevertheless, we see that AspectJ will not provide such features in the near future. Consequently, we do not think that for industrial applications it will be possible in the near future to have join point selections across local boundaries.

The adaptation of the web representation is also not possible to be handled by AspectJ, because the web representation was build up using velocity templates. Consequently, it is not valid to say that the application of aspect-oriented techniques guarantees complete separation of all security related concerns: Because of the underlying techniques being used there is still crosscutting code that cannot be modularized.

6 Related Work

In the PhD thesis by Bart De Win [20] the modularization problem for application-level security is discussed. The discussion is based on studying how aspect-oriented techniques can optimize the modularization of application-level security requirements. The thesis contains discussions about interceptor and weaving approaches for implementing security-related concerns in different kinds of application. One is a FTP server, another is Personal Information Manager. The author concludes that the security concern is highly crosscutting. Although instance-based authorization mechanisms were considered during the case study, the architecture of the example application does not correspond to the architectural constraints used in this paper. Consequently, it is not directly possible to transfer the results of [20] to the J2EE context. Also, no quantitative comparison between the pure object-oriented and aspect-oriented versions of the same concerns was provided.

[19] analyzes software security problems and tries to answer the question whether security can be separated from other aspects of an application and what software engineering techniques should be used in order to achieve a maximum separation. The
work discusses what kinds of problems exist, how they can be solved using aspect-oriented techniques, and what advantages and disadvantages the aspect-oriented solution has. The paper concludes with two important benefits of the aspect-oriented approach in software security – adaptable security mechanisms and unanticipated evolution advantages. However the paper is based on a theoretical discussion and gives a qualitative analysis. It does not directly refer to the language specific characteristics or any architectural constraints.

In [18] John Viega et al describe an aspect-oriented extension to the C programming language that has significant benefits in the realm of security. The authors describe a simplified aspect-oriented pre-compiler that weaves the advices in to the original C program. The primary example of the security flaw is considered to be a buffer overflow that is outside of the scope of this paper.

A case study made by Gao et al. [4] describes the implementation of the role-based access control (RBAC) and its variations on CORBA access control. The described implementation covers only a part of the security issues – the authorization, and leaving authentication, confidentiality and non-repudiation features implemented inside CORBA platform. Also the work assumes a class-based authorization, which is in our concrete application not applicable.

7 Conclusion

In this paper we argued for the need to study security as a separate concern in the J2EE context. The main motivation is to understand whether aspect-orientation has a benefit in comparison to conventional composition techniques for this special concern in the given architectural context.

We introduced a blog application and designed and implemented it in a J2EE environment without considering any security issues. Then, we identified some security requirements. Afterwards, we implemented the security concerns using conventional composition techniques in the J2EE environment as well as using the aspect-oriented language AspectJ. For each version we explained exemplary the underlying design decisions and implementation issues for each facet of the security concerns. Then, we expressed the overall impact of each implementation strategy for different categories in terms of lines of code.

The study has shown that security concerns are highly crosscutting in the J2EE context. The implementation of such concerns using conventional composition techniques required an intensive adaption of the core functionality. As a consequence, the original application becomes rather hidden within a non-functional code. The aspect-oriented solution provided a much clearer modularization of the concerns and permitted to keep almost all classes from the original application as is. However, the aspect-oriented solution did not turn out to be completely orthogonal to the original code – it was still necessary to change a number of class definitions or velocity template files.
We regard the main contribution of this study in setting three applications in direct relationships to each other. This permits to study the impact of a certain concern on the original application using conventional as well as aspect-oriented techniques. Furthermore, the study considers all elements of the software (including non-java files). This permits to give a more objective view on the impact of security related concerns on the resulting software.

We are aware of the fact that number of lines of code is a very simplified index for expression of such an impact. Other indexes like proposed in [20] might help in arguing for the aspect-oriented solution, but might not help arguing why the object-oriented solution is rather inappropriate. In our concrete application LOC turned out to be sufficient, because the number of lines of code already dramatically differs in the pure J2EE and the aspect-oriented implementation. We are also aware that the original blog application is rather a very simple application. However, since the main intention was to study security concerns in the J2EE context this small application turned out to be useful, because all underlying design decisions can be easily understood and explained.

All in one we can conclude that in a J2EE context security related concerns lead to crosscutting code that is rather unacceptable, because the concerns tackle almost all existing modules. The aspect-oriented implementation in AspectJ provides a better modularity of the corresponding code but still is not able to separate all concern specific code into corresponding modules.

References


GSEM 2005