Original Article

Applying Aspect Oriented Programming on Security

Mohammad Khalid Pandit*[1], Azra Nazir[1] and Arutselvan M[2]

[1]Department of computer Science and engineering, National institute of technology Srinagar, India
[2]MGR University Chennai, India

Address for Correspondence

Department of computer Science and engineering, National institute of technology Srinagar, India.
E-mail: khalidpandit@gmail.com

ABSTRACT

The problem of code scattering and tangling is very common among sizeable applications. These result in crosscutting concerns.. The issues that are related to security are particularly severe. Mechanisms are being developed to deal with different concerns separately. An interesting case of this separation is security. The implementation of security mechanisms are usually scattered all over the code and tangled with the core functionality of the application. This results in unmanaged code with high risk. Aspect-oriented programming (AOP) promises to tackle the problem of crosscutting concerns by offering several abstractions that help to reason about and specify the concerns individually. Aspect-oriented programming is an emerging programming paradigm that seeks ways to modularize software systems. Modularizing involves separating and localizing the different concerns. State-of-the-art software techniques already support separating concerns, for instance by using method structuring, object-oriented programming and design patterns. However, these techniques are inadequate for more complex modularization problems (security). Aspect-oriented programming is an approach that provides more innovative modularization techniques i.e. it helps to minimize these risks by eliminating the tangling and scattering of the code.

Keywords: Programming languages, Aspect oriented programming, Security, Separation of concerns, Modularization.

INTRODUCTION

The principle of separation of concerns proposes encapsulating features into separate entities in order to localise changes to them and deal with one important issue at a time.

While software development the security should never be considered as the minor issue and security should never be added to an application as an afterthought because it leads to bugs and vulnerabilities[1]. The security should be considered as an

issue in each and every phase in software development right from requirements gathering to final implementation. It is relatively easier to take security into account in the initial phases of development like requirements gathering and analysis. But it becomes harder as the development reaches higher and more complicated stages because not only application but the security mechanism also becomes more complex. The major problem is the interaction between the functionality of the application and how security policy should work[2]. At the root of this problem lies the structural mismatch between the application logic and the required security solution. This security mismatch can be eliminated if the application logic and security and every other concern is properly modularized. 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 inadequate for complex modularization problems[13]. E.g Object oriented programming paradigm separates concerns in an intuitive manner by grouping them into objects. However, object oriented paradigm only helps in modularizing concepts that easily map to the objects, but it is not good at separating concerns[4]. For example it is difficult to model security in object oriented paradigm, while we can write a central security manager for the application, and explicit calls to be made to the security manager from every spot where security is needed. Unfortunately if the important call to security manager is forgotten from a point in the application it causes a security leak at that point. i.e. forgetting to trigger security checks at sensitive points in an application can lead to hard-to-spot security holes. Aspect oriented programming can solve this problem by allowing security concerns to be specified modularly and main application in a uniform way.

Aspect oriented programming is a new programming paradigm that explicitly promotes the separation of concerns[9]. In the context of security this would mean that the main program should not need to encode security information[4], instead it should be moved to separate independent piece of code. This helps to reduce the tangling and scattering of security related code in the application. Modern programming techniques that support separation of concerns like object oriented programming, method structuring, encapsulation etc 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 final result is that conventional modularization techniques are unable to entirely modularize *crosscutting concerns*.

At large, every software application has two types of concerns associated with its operation i.e. primary concern and secondary concern. Usually the primary concerns in an application do not crosscut with other concerns; it is the secondary concerns which crosscut the application[12]. E.g. consider the case of File access. The primary concern in this operation is the updation or deletion of the file, while as the secondary concern is the security related to the operation. The security concern crosscuts the application and the code related to security is scattered with other concerns. This causes the security of the application precarious.

Aspect oriented programming is the answer to this problem. It has constructs to declare how modules crosscut one another. In this paper we use AspectJ, an aspect oriented extension of java[10]; that helps

dealing with crosscutting at implementation level.

The problems caused by crosscutting concerns in the implementation of software are well known, and are the raison d'^etre of the aspect-oriented software development community[3,5]. In the particular case of security related applications, there are at least three specific issues:

(1) It is not easy to change the current access control implementation (e.g., to change the kind of security policies being enforced) because it is not modularly defined.

(2) Programs that do not take security into account cannot be made security aware without directly modifying them.

(3) Forgetting to trigger access control checks at sensitive points in an application can lead to hard-to-spot security holes.

Motivation

Separation of concerns reduces system complexity caused by mixing crosscutting concerns, which are aspects of a system that affect other concerns. Secure software systems can be developed by separating application and security concerns with the goal of making these systems more maintainable and reusable[6]. By careful separation of concerns, the security requirements are captured separately from the application requirements. In the design, security concerns are modelled in security components separately from the application components as well.

An aspect plays an important role to separate security code from application code in the implementation. An aspect can be described as a combination of four integral parts: the aspect itself, a join point (s), a pointcut (s), and advice[6,3]. These concepts are crucial to creating an implementation model with separation of concerns from design models in aspect oriented programming. Though definitions may vary, an aspect is generally thought of as a feature of a system, which is scattered at multiple points throughout the system. Aspects are commonly used to represent crosscutting concerns that are separated from the core business logic of a system. For example, imagine a File access application where a user can implement the operations like delete, update or add. The operation depends upon the permissions of the user given by the security system, ie some users may be authorized to delete or update a particular file while other users may not be authorized. Core business logic for this application system would be the methods that involve delete or update of file chosen by the user, whereas concerns separated from business logic would include all security concerns such as authentication and access control. Thus security concerns can be modelled with both authentication and access control as separate aspects of the system, because they are not directly involved with core business logic. (See figure 1.)

Introduction to aspectj

Aspect-oriented programming is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns[3,7]. AOP forms a basis for aspect-oriented software development. AspectJ is the aspect oriented extension of the java language. All valid Java programs are also valid AspectJ programs, but AspectJ also allows programmers to define special constructs called *aspects*. Aspects can contain several entities unavailable to standard classes[7]. These are:

Inter-type declarations

Allow a programmer to add methods, fields, or interfaces to existing classes from within the aspect.

```
aspect VisitAspect {
 void Point.acceptVisitor(Visitor v) {
   v.visit(this);
 }}
```

Pointcuts

Allow a programmer to specify join points (well-defined moments in the execution of a program, like method call, object instantiation, or variable access). All pointcuts are expressions (quantifications) that determine whether a given join point matches. For example, this point-cut matches the execution of any instance method in an object of type Point whose name begins with set:

```
pointcut set() : execution(* set*(..) ) && this(Point);
```

Cross-cutting concerns

Even though most classes in an OO model will perform a single, specific function, they often share common, secondary requirements with other classes. For example, the security is a concern which spans all over the application ie in our example the call to security manager is done in both methods (delete and update).

Advice

This is the additional code that you want to apply to your existing model. In our example, this is the deletion code that we want to apply whenever the thread enters or exits a method.

Aspect

The combination of the pointcut and the advice is termed an aspect. In the example, we add a secure aspect to our application by defining a pointcut and giving the correct advice.

In AspectJ we can use the pointcut-advice (PA) model[5] for aspect-oriented programming, crosscutting behavior is defined by means of pointcuts and advice. Execution points at which advice may be executed are called (dynamic) join points. A pointcut identifies a set of join points, and a piece of advice is the action to be taken at a join point matched by a pointcut. An aspect is a module that encompasses a number of pointcuts and pieces of advice. Following is the shape of an aspect in AspectJ, which follows the PA model:

```
aspect AspectExample {
pointcut pc(): . . . //predicate selecting join points
before(): pc() {
//action to take before selected join point execution
```

Figure 2 shows two implementations of this example: an ordinary object-oriented implementation in Java, and an aspect-oriented implementation in AspectJ. The key difference between the implementations is that in the AOP version the security behaviour is implemented in an aspect, whereas in the non-AOP code it is scattered across the methods of update and delete.

In the aspect *secure*, the first member declares a pointcut named cross (). This pointcut identifies certain join points in the program's execution, specifically the execution of the update and delete methods in Deletion. (See figure 2.)

```
Public class deletion
{
  public void delete()  {
        // Permission checking (authorization)
        // Logging
        // Checking for the authentic User
        // Actual Deletion Logic comes here
   }
    public void update()  {
        // Permission checking (authorizing)
        // Logging
        // Checking for the authentic User
        // Actual updation Logic comes here
  } }
```

Typical implementation of delete and update methods.

The above pseudo code represents the typical implementation of update and delete methods in our File deletion example. As shown in fig. 3 apart from actual implementation code in that method all other are the cross cutting concerns (secondary concerns) which cause the scattering and tangling of the code. E.g. permission checking, logging, checking for authentic user are the cross cutting concerns. The aspect oriented programming removes these kinds of concerns by defining the cross cutting concerns as aspects. Take the example of checking for authentic user, AOP defines this as an aspect:

```
Aspect Authentication {
Pointcut cross() : execution (void Deletion.update(File)) ||
execution (void Deletion.delete(File));
Before() {
if (args[0] instanceof User)
{//user has access rights
User user = (User)args[0];
// Authenticate if he/she is the right user }
```

We can use the concept of permission aspects and restriction aspects to remove the cross cutting concerns like authentication in file access problem. Deploying Permission aspect[5] is equivalent to performing the explicit invocation to Security Manager. Check Permission in delete or update method. However, the fundamental advantage of the aspect-oriented approach is that explicit calls to Security Manager. Check Permission are no longer necessary. (See figure 4.)

```
Aspect Permission {
pointcut cross() : execution (void Deletion.delete(File)) ||
execution (void Deletion.update(File));
Before() {
if (args[0] instanceof User)
{ SecurityManager.checkPermission(
newFilePermission(this.path,
FILEACCESS_ACTION)
User user = (User)args[0];
// Authenticate if he/she is the right user }
```

Another kind of aspects are needed based on a different mechanism for access control enforcement: restriction aspects. A restriction aspect, instead of invoking Security Manager. Check Permission in its advice, throws an exception as soon as it sees the resource access its pointcut identifies.

```
Aspect Restriction {
pointcut cross() : execution (void Deletion.delete(File)) ||
execution (void Deletion.update(File));
before(){
if(User.Id.equals( “Invalid”)) {
throw new AccessControlException()||
srcurityException();
}
}}
```

CONCLUSION

This paper outlines an approach for implementing complex systems by separating application and security concerns. The goal of this research is to reduce overall system complexity and increase modularity and the reusability of certain concerns in application systems. This goal is imagined through the careful separation of crosscutting security concerns from business logic in the software development. In this paper, we used the Java programming language and AspectJ extension to make this separation of concerns a reality during implementation of a File deletion example.

REFERENCES

1. International standards ISO IS 15408 common criteria for information technology security evaluation (parts 1-3), version 2.1, September 2000.

2. Bart De Win, Bart Vanhaute, and Bart De Decker "*How Aspect oriented programming can help to building secure software*?".
3. AspectJ Website. http://www.aspectj.org/.
4. J. Viega, J. Bloch, and P. Chandra, "*Applying Aspect-Oriented Programming to Security*", *Cutter IT Journal*, vol. 14, no. 2, pp. 31-39, Feb. 2001.
5. Rodolfo Toledo, Angel Nu´n˜ ez, _ Eric Tanter "*Aspectizing java access control*".
6. Chase Baker Michael Shin "Mapping of Security Concerns in Design to Security Aspects in Code".
7. Aspect oriented programming, Wikipedia page http://en.wikipedia.org/wiki/Aspect-oriented_programming.
8. Taeho Kim and Hongchul Lee "Establishment of a Security System using Aspect Oriented Programming".
9. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J." Aspect-oriented programming".
10. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. "An overview of aspectj".
11. Azzam Mourad, Marc-Andr´e Laverdi` ere and Mourad Debbabi "Towards an Aspect Oriented Approach for the Security Hardening of Code".
12. Mohammad Khalid pandit "Developing secure software using aspect oriented programming" *IOSR-JEC* vol 10, issue 2, mar, apr 2013.
13. Bart De Win, Joosen, and Frank Piessens "*Developing Secure Applications through Aspect oriented programming.*

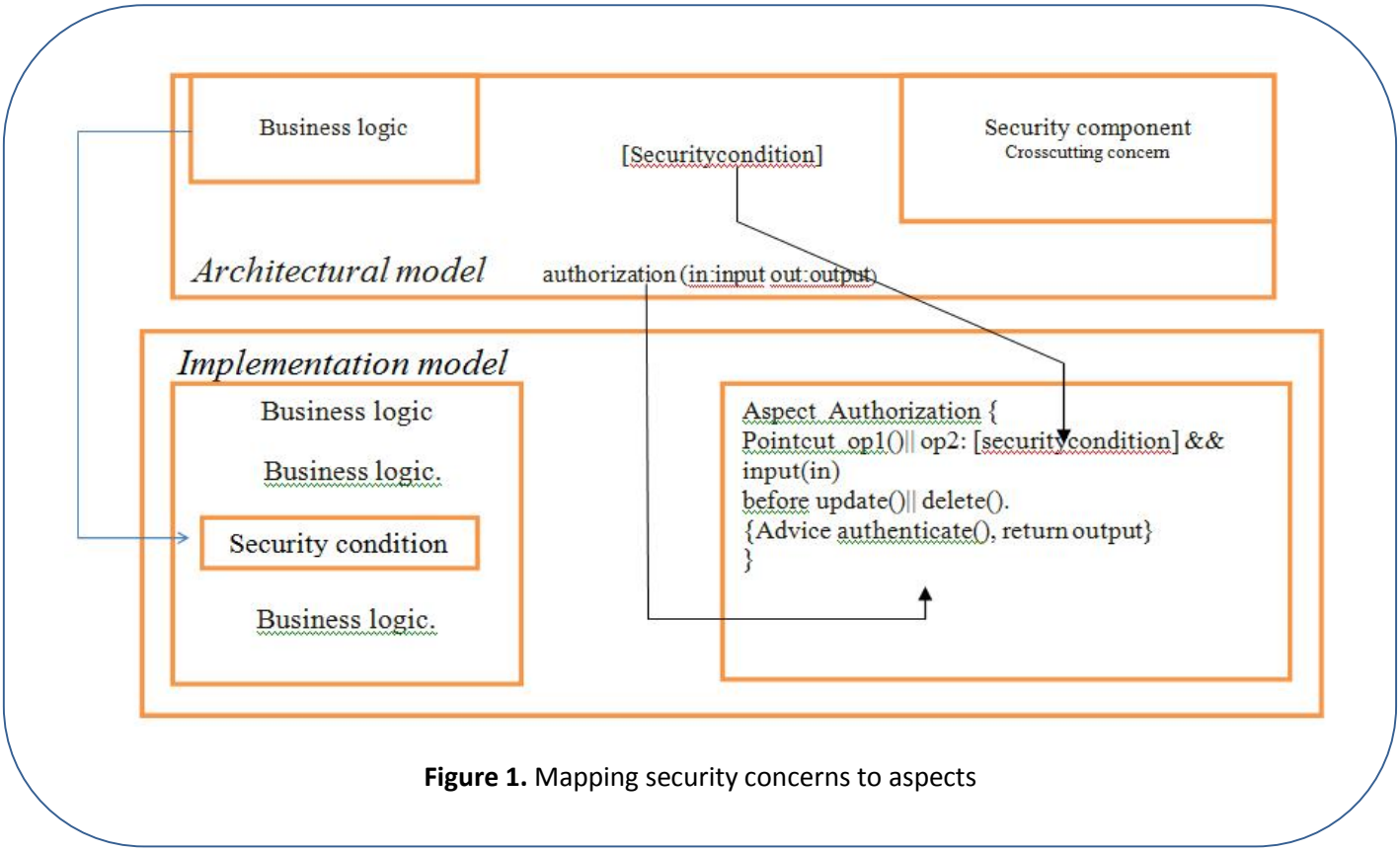

Figure 1. Mapping security concerns to aspects

Example

```
Class Deletion

{

Public void update()    {return ac;}

Public void delete  (File path) {

SecurityManager.checkPermission(
   new FilePermission(this.path,
FILE_DELETE_ACTION) );
//actual  deletion  goes here ...
}

Public void update (File path) {

SecurityManager.checkPermission(
   new FilePermission(this.path,
FILE_UPDATE_ACTION) );
//actual  Updation goes here ...
}}
```

```
Class Deletion
{
Public void update()    {return File;}

Public void delete  (File path) {

//actual detetion  code

public void clean()  {

  AccessController.doPrivileged(new PrivilegedAction(){
    public Object del() {
      for(String path: path){
        new cleanfile(path).del();
      }}
Public void update (File path) {

      for(String path: path){
        new updatefile(path).upd();
          //actual Updation code}};

Aspect secure {
Pointcut cross () : execution  (void Deletion.delete(File))
      || execution (void Deletion.update(File))
before() returning: cross() {
SecurityManager.checkPermission(
  new FilePermission(this.path,  FILEUPDATEACTION) );}}
```

Figure 2. Java and AspectJ implementation of file deletion example

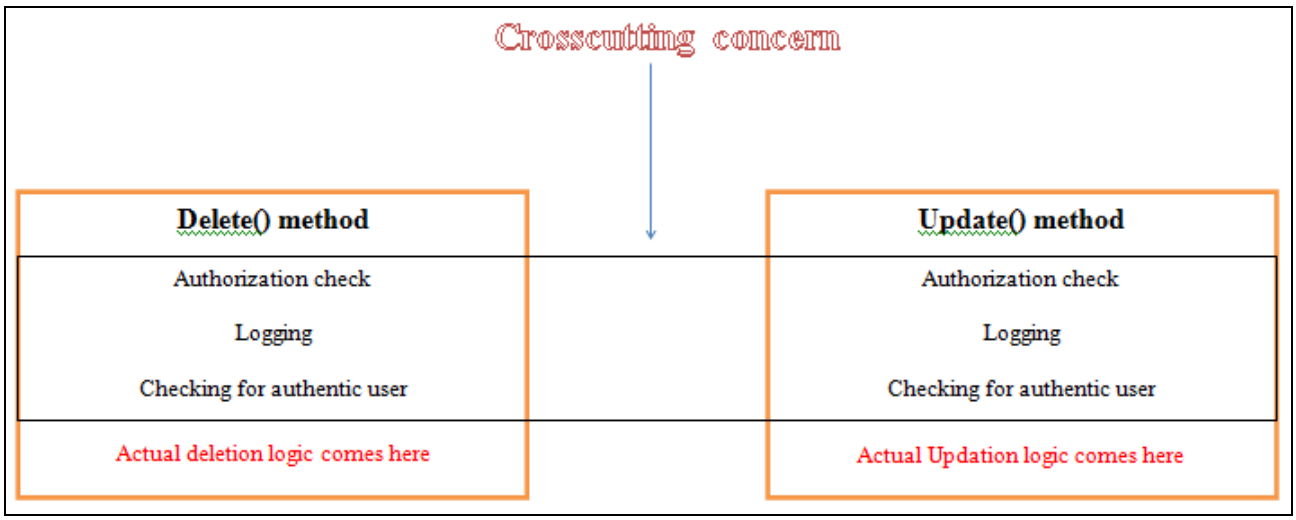

Figure 3. Implementation of cross cutting concerns

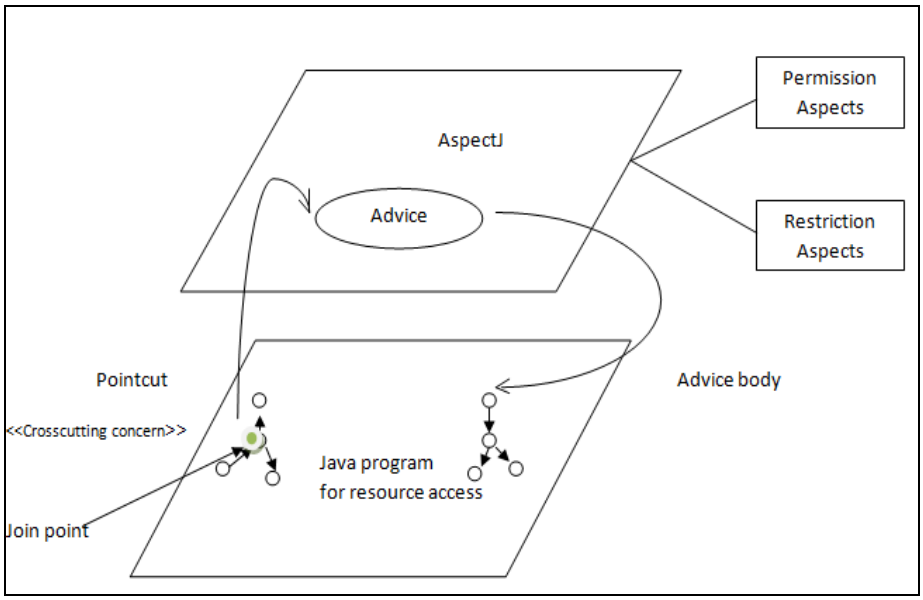

Figure 4. Architecture diagram