Modularizing Security concerns using Aspect-oriented Programming

Ms.K.Santhis, Dr.G.Zayaraz

Research Scholar/CSE, Professor/CSE Pondicherry Engineering College, Puducherry, India.
gzayaraz@pec.edu

Abstract

A software system is composed of a number of concerns, each built by one or several functional requirements and/or non-functional requirements. In software system development, one of the essential principles is separation of concerns. Generally separation of concerns is used to break the software system into several modules that overlap minimally. Nevertheless, there are some particular concerns that cannot be located in a single module; these are referred to as crosscutting concerns. Aspect-oriented programming (AOP) provides techniques for managing crosscutting concerns into a single manageable module, referred to as aspect. AOP is used to solve many problems, such as tangling and scattering representations. However, the identification and specification of crosscutting concerns, and considering them as aspect, is challenging. These challenges are particularly severe in the case of security aspects. AOP can help to reduce these risks by removing the tangling and scattering of the code. The object-oriented programming paradigm separates concerns in an intuitive manner by grouping them into objects, though this paradigm is only effective at separating out concepts that easily map to the objects, not at separating concerns. A security policy called Same Origin Policy which restricts the amount of damage that could happen due to cross origin interaction while the injection of aspects. It utilizes a new data structure called Lazy splay trees for detection of hazardous aspects.

Keywords: Aspect-oriented programming, Lazy splay, Same Origin Policy, Scattering. Separation of concerns, Tangling.

1. Introduction

Today’s world in Internet, it is essential to use in any applications in a well secured manner. However, the development of a secure application in an openibly-distributed environment is a far from straightforward task. In software development, security should never be considered a minor issue, and security should never be added to an application as an afterthought, as this leads to bugs and vulnerabilities [1], [2]. Security should be considered as an issue in each and every phase in software development, from requirements gathering to final implementation. It is relatively easier to take security into account in the initial phases of development, such requirements gathering and analysis. However, it becomes harder as the development reaches higher and more complicated stages, as not only the application but the requisite 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 removed if the application logic and security and each concern is properly modularized. State-of-the-art software system techniques already support separating issues, for example by victimization methodology structuring, clean object-oriented programming and style patterns. These techniques are inadequate for more complex modularization problems [11], [13], [16]. The object oriented programming paradigm typically separate concerns in an intuitive manner by grouping them into objects. However, object oriented paradigm is simply smart at separating out ideas that simply map to the objects, however it’s not smart at separating considerations concerns [4] For example it is difficult to model security in object oriented paradigm, while we can write a central security manager for the

* Corresponding author. Tel: E-mail: santhikrishnan@pec.edu
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 (AOP) can solve this problem by allowing security concerns to be specified modularly and main application in a uniform way.

Aspect oriented programming (AOP) could be a new programming paradigm that expressly promotes the separation of concerns [9]. Within the context of security this is able to mean that the most program mustn't have to be compelled to inscribe security data [4], instead it ought to be emotional to separate freelance piece of code. This reduces the tangling and scattering of security connected code within the application. The most important reason for this drawback is the inherently focused focus of these techniques on one view of the problem; they lack the ability to approach the problem from different viewpoints concurrently. It is evident that the conventional modularization techniques are unable to fully separate crosscutting concerns.

In general, each software package application has 2 sorts of considerations related to its operation i.e. primary concern and secondary concern. Typically the first considerations in associate degree application don't crosscut with different considerations; it's the secondary concerns that crosscut the appliance [12]. E.g. consider the case of withdrawal of cash from the ATM, the first concern during this operation is that the withdrawal of the money whereas because the secondary concern is that the security associated with the operation, the security concern crosses the appliance and causes the security connected code to be scattered with different concerns. This causes the security of the appliance precarious.

Aspect orientated programming is that the answer to the present drawback, it's constructs to declare however modules crosscut each other. During this paper we have a tendency to use AspectJ, an aspect orientated extension of java [10]; that helps addressing crosscutting at implementation level.

The problems caused by crosscutting concerns in the implementation of software are well known, and are the raison d'être of the aspect-oriented software development community [3][5]. In the particular case of security related applications, there are at least three specific tight spot: 1) it's dangerous to alter the present access management implementation (e.g., to alter the type of security policies being enforced) as a result of it's not modularly defined; 2) programs that don't take security under consideration can't be created security aware while not directly modifying them.3 forgetting to trigger access control checks at sensitive points in an application can lead to hard-to-spot security holes.

2. 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],[12]. By careful separation of concerns, the security requirements are captured separately from the application requirements. In the design, security concerns are modeled 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 [3],[6]. 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 an ATM banking transaction where a user can implement any of the operations like money withdrawal, deposit, transfer etc. the operations also depends upon the permissions of the user given by the security system. Core business logic for this application system would be the methods that involve withdrawal and deposit of the money chosen by the customer, whereas concerns separated from business logic would include all security concerns such as authentication and access control. Thus security concerns can be modeled with both authentication and access control as separate aspects of the system, because they are not directly involved with core business logic. Furthermore, the aspect itself is composed of smaller components such as the point cuts and advice [3],[4]. A point cut refers to a set of join points within a system, which are well defined points of execution such as method calls, method executions, or object initializations. This point cut is used to determine at what point in the system advice needs to be executed. Referring to the ATM transaction example, the authentication aspect should have a point cut before execution of the business logic method that shows a list of operations to the customer.

This ensures that the user is valid before the system displays any operations to the customer. Also, the Access Control aspect could have a point cut around the method in which the customer selects an operation to perform. This also ensures a customer has access permission (or man balance) to carry out the operation that she or he has chosen (fig 1).

3. Introduction to aspect

Aspect-oriented programming (AOP) could be a programming paradigm that aims to extend modularity by permitting the separation of cross-cutting concerns [3][7],[8]. AspectJ is the aspect oriented extension of the
java language. All valid Java programs are also valid AspectJ programs; however Aspect J conjointly permits programmers to outline special constructs referred as aspects. Aspects have a number of 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 [7][8][11].

```java
aspect VisitAspect
{
  void Point.acceptVisitor (Visitor v)
  {
    // visitor code
  }
}

pointcuts — allow a programmer to specify points. All pointcuts are expressions determine whether or not a given join point matches. For an example, this point-cut matches the execution of any instance method in an object of type Point whose name begins with set:[14][16]

advice — allows a programmer to specify code to run at a join point matched by a pointcut. The actions can be performed before, after, or around the specified join point[14][16]. At this point, the advice restore the display
every time something on Point is set, using the pointcut declared above:
```java
    after (): set()
    {
        Display.update();
    }
```

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. The point where advice is executed is referred as 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 [8],[11],[15]. An aspect is a module that encompasses a number of pointcuts and pieces of advice. The shape of an aspect in AspectJ, which follows the pointcut-advice model is given below:

```java
aspect AspectExample
{
    pointcut pc(): . . . //predicate selecting join points before (): pc() {
        //action to be taken before the execution of selected join point
    }
}
```

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 update behaviour is implemented in an aspect, whereas in the non-AOP code it is scattered across the methods of withdraw and deposit. In the Updatetr aspect, the first member declares a pointcut named cross(). This pointcut identifies certain join points in the program’s execution, specifically the execution of the withdraw and deposit methods in Transaction, as well as transfer method defined in Exchange is shown in Table 1.

![Crosscutting concerns](image)

**Fig 3. Identification of Crosscutting concerns**

<table>
<thead>
<tr>
<th>Class Transaction</th>
<th>Class Transaction</th>
</tr>
</thead>
</table>
| ```java
    int amnt, ac;
    Float bal;
    public int getac() { return ac; }
    public float deposit(int amnt)
    {
        this.bal = this.bal + amnt;
        Display.Update();
        return bal;
    }
    public float withdrawal (int amnt)
    {
        this.bal -= amnt;
        display.Update();
        return bal;
    }
``` | ```java
    int amnt, ac;
    Float bal;
    public int getac() { return ac; }
    public float deposit(int amnt)
    {
        this.bal = this.bal + amnt;
        return bal;
    }
    public float withdrawal (int amnt)
    {
        this.bal -= amnt;
        return bal;
    }
``` |

<table>
<thead>
<tr>
<th>Class Transfer</th>
<th>Class Exchange</th>
</tr>
</thead>
</table>
| ```java
    private Transaction T1, T2;
    public Transaction getac1() { return T1; }
    public Transaction getac2() { return T2; }
    public void transfer (int amnt)
    {
        T1.bal += amnt;
        T2.bal += amnt;
        Display.update();
    }
``` | ```java
    private Transaction T1, T2;
    public Transaction getac1() { return T1; }
    public Transaction getac2() { return T2; }
    public void transfer (int amnt)
    {
        T1.bal += amnt;
        T2.bal += amnt;
    }
``` |

<table>
<thead>
<tr>
<th>AspectUpdate</th>
<th>After() returning: cross()</th>
</tr>
</thead>
</table>
| ```java
    pointcut cross (): execution (void Transaction, deposit (int))
    || execution (void Transaction, withdrawal (int))
    || execution (void Exchange, transfer (int));
    after () returning: cross();
    display.Update();
``` | ```java
    ``` |
The above pseudo code represents the typical implementation of withdraw and deposit methods in our ATM transaction example. As shown in fig 3 apart from actual implementation code in that method all other are the cross cutting concerns which cause the scattering and tangling of the code. E.g. transaction management, logging, checking for privileged 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 privileged user in AOP, this concern can be defined as an aspect:

Aspect Authentication {
  Pointcut cross() : execution (void Account.deposit(int)) ||
  execution (void Account.withdraw(int));
  Before(){
    if (args[0] instanceof User) {
      User user = (User)args[0];
      // Authenticate the right user
    }
  }
}

We can use the concept of permission aspects and restriction aspects to remove the cross cutting concerns like authentication in Withdraw and Deposit methods. Deploying Permission aspect [5],[9] is equivalent to performing the explicit invocation to SecurityManager.checkPermission in withdrawal or deposit method. However, the fundamental advantage of the aspect-oriented approach is that explicit calls to SecurityManager.checkPermission is not compulsory.

Aspect Permission {
  pointcut cross(): execution (void Account.deposit(int)) ||
  execution (void Account.withdraw(int));
  before(){
    if (args[0] instanceof User) {
      User user = (User)args[0];
      // Authenticate the right user
    }
  }
}

Another kind of aspects are needed based on a different mechanism for access control enforcement is referred as restriction aspects. Instead of invoking SecurityManager.checkPermission in its advice, restriction aspects throws an exception as soon as it sees the resource access its pointcut identifies.

Same-origin policy approach is utilized to inflict the policy model at run time which permits aspects originating from the same location like a combination of scheme, hostname and port number. The weaver supplements woven software with logic to maintain the permission state of the software. As such, solely the weaver is altered and no modification of the virtual machine or language linguistics is needed.

Fig 4. Same origin policy approach

Listing 1. Hazardous AspectJ Inspection
Listing 1 illustrate that the mechanism of methodology inception will be used to disable the java security and intrinsically disable all Java security features.

```java
Rebalance (Node parent, Node node)
{
    nodePlusLeftCount = node.selfCnt+node.leftCnt;
    parentPlusRightCount =
    parent.selfCnt+parent.rightCnt;
    nodeRightCount = node.rightCnt;
    //decide whether to perform zig-zag step
    if (nodeRightCount >= parentPlusRightCount)
    {
        Node grand = parent.parent;
        ZigZag (grand, parent, node, rightChild);
        parent.leftCnt = rightChild.rightCnt;
        rightChild.rightCnt = rightChild.leftCnt;
        rightChild.leftCnt += parentPlusRightCount;
        rightChild.leftCnt += nodePlusLeftCount;
    }
    else
    //decide whether to perform zig step
    if (nodePlusLeftCount > parentPlusRightCount)
    {
        Node grand = parent, parent;
        Zig (grand, parent, node, node.right);
        parent.leftCnt = node.rightCnt;
        node.rightCnt += parentPlusRightCount;
    }
}
```

To construct the counting based lazy splay trees, consider aspects with the corresponding node stickers as in Table II.

<table>
<thead>
<tr>
<th>No.</th>
<th>Aspects</th>
<th>Node sticker</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Exception Handling</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Persistence</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>security</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Monitoring</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>Logging</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>Synchronization</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>Transaction processing</td>
<td>7</td>
</tr>
</tbody>
</table>

Injection of Synchronization, Persistence, Transaction, Exception, Monitoring, Logging and Security aspects are shown in the Fig 6.
If noderightcounter > Parent plusrightcounter perform Zig-Zag rotation and update the counters as follows.

- parent.leftCnt = rightChild.rightCnt;
- node.rightCnt = rightChild.leftCnt;
- rightChild.rightCnt += parentPlusRightCount;
- rightChild.leftCnt += nodePlusLeftCount

In order to reduce the time complexity needed, the counter based splay trees are used to verify whether the injected aspects belong to the same origin or not.

4. Conclusion

This paper outlines an approach to 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 expected through the careful separation of crosscutting security concerns from business logic in the software development. In this paper, we used the same origin policy to make this separation of concerns a reality during implementation of an ATM transaction example.

References