IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p- ISSN: 2278-8727Volume 10, Issue 2 (Mar. - Apr. 2013), PP 28-34
www.iosrjournals.org

Developing secure software using Aspect oriented programming

Mohammad Khalid Pandit
Dept. of Computer Science and Engineering Arunai Engineering College

Abstract*: Aspect oriented programming (AOP) is the programming paradigm that explicitly promotes the separation of crosscutting concerns. Some concerns crosscut the sizable application resulting in code scattering and tangling. These concerns are particularly severe in case of security related applications. The security of these applications can become compromised when the security related concerns are scattered and tangled with other concerns. The object oriented programming paradigm sometimes separate concerns in an intuitive manner by grouping them into objects. However, object oriented paradigm is only good at separating out concepts that easily map to the objects, but it is not good at separating concerns. Aspect oriented programming is the promising approach to improve the software development process and can tackle this problem by improving the modularity of crosscutting concerns.*

Keywords— *Programming languages, Aspect oriented programming, Security, Separation of concerns.*

I. INTRODUCTION

Aspect oriented programming (AOP) is a new programming paradigm that explicitly promotes the separation of concerns. In the context of security this would mean that the main program should not need to encode security information, instead it should be moved to separate independent piece of code. This reduces the tangling and scattering of security related code in the application. State-of-the-art software techniques already support separating concerns, for instance by using method structuring, clean object-oriented programming and design patterns. However, these techniques are insufficient for more complex modularization problems. A major cause for this limitation is the inherently forced focus of these techniques on one view of the problem; they lack the ability to approach the problem from different viewpoints simultaneously. The net result is that conventional modularization techniques are unable to fully separate *crosscutting concerns.*

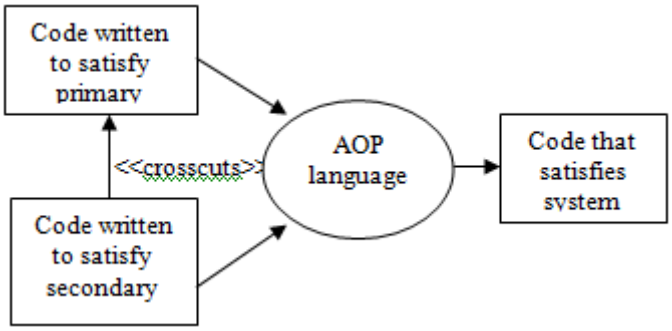
Aspect-oriented programming (AOP) is an approach that provides more advanced modularization techniques. The main characteristic of this technology is the ability to specify both the behavior of one specific concern as well as how this behavior is related to other concerns (the *binding*). In fact,

AOP has become a general term to denote several approaches to provide such a development functionality. One prominent tool in this space is AspectJ. AspectJ extends Java with mechanisms for expressing advanced modularization capabilities. In AspectJ, a unit of modularization is called an *aspect* and a unit of binding is a *pointcut*.

Security should never be considered a minor issue when developing software. Adding security after the development is always a bad idea, and will probably lead to bugs and vulnerabilities. Security should be dealt with in each phase of development process, from requirements gathering to testing and final deployment.

During the initial phases of development (requirements gathering) it is relatively easy to take security considerations into account. It can be rather dealt independent of the application. However, later in the development cycle it becomes harder to handle the security requirements. As the development process goes on not only the application but also the security mechanisms become more complicated. The real problem lies in the interaction between the application functionality and what the security needs to work properly. At the base of this problem is the modularity issues between the application and the security mechanism, which causes the security related code to be scattered and tangled with other concerns of the application which can compromise the security of the application.

In general, 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. E.g. consider the case of withdrawal of money from the ATM. The primary concern in this operation is the withdrawal of the money while as the secondary concern is the security related to the operation. The security concern crosscuts the application and causes the security related code to be 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; 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. In the particular case of security related applications, we can mention 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; and 3) forgetting to trigger access control checks at sensitive points in an application can lead to hard-to-spot security holes. If permission checks are not modularized, it is hard to reason about them globally.

II. MOTIVATION

Experience has shown that developers are not very good at writing secure software. Even the security aware people find it much easier to write their program and then later go back and try to "bolt on" security as an afterthought. Actually this approach is flawed as the security needs to be taken into consideration from the very beginning of the development process. There happens to be a fundamental conflict between what works well and the way developers work.

From the viewpoint of secure software there are different responsibilities in an application. In case of security we can distinguish between three different categories.

1) Code involving core functionality (business logic).
2) Code implementing basic security operations (code unrelated to business logic)
3) Code that specifies how and where to use security in the application.

The third responsibility actually constitutes the relationship between the business logic and security of the application.

If we apply this logic to the example of online banking system, there is the code that implements how to deposit the money, withdraw, balance checking, money transfer from one account to another, it is a part of business logic of the application and belongs to the core functionality part. Also there is code that takes care of security of the transactions. This kind of code can be reused for various types of applications where similar kind of security is needed. This code belongs to the second category. And finally some code in the application that is responsible for activating the security mechanisms at the right place and the right time i.e every time a transaction is made. This piece of code links security to the application and therefore part of third category.

According to the stated logic the security should be considered seriously in every developmental phase as uncareful development can lead to security problems.

If we consider the traditional software development, the secure application development is not full proof due to the very design of the development. There are modularity issues which causes the code to be scattered and tangled within the application and causes the security concerns. In the case of access control in java the access control is the cross cutting concern and causes modularity issues. The code of the access control is scattered aver all other classes and is tangled with other concerns. Fig 1 shows the access control checks in standard distribution of java 1.5. The white bars represent the length of the class in terms of lines of code while as black bars represent the length of access control code in these classes.

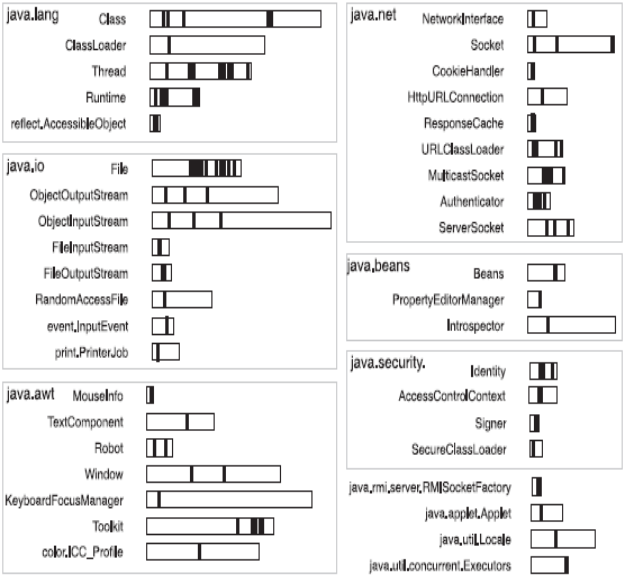


Fig 1.Access control checks in standard distribution of java 1.5

III. INTRODUCTION TO ASPECTJ

Aspect-oriented programming (AOP) is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns. 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. 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);
```

- **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. Here, the advice refreshes the display every time something on Point is set, using the pointcut declared above:

```
after () : set() {
  Display.update();
}
```

The various concepts that are very important in AspectJ are:

1) 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, we may want to add logging to classes within the data-access layer and also to classes in the UI layer whenever a thread enters or exits a method. Even though each class has a very different primary functionality, the code needed to perform the secondary functionality is often identical.
2) Advice: This is the additional code that you want to apply to your existing model. In our example, this is the logging code that we want to apply whenever the thread enters or exits a method.
3) Pointcut: This is the term given to the point of execution in the application at which cross-cutting concern needs to be applied. In our example, a pointcut is reached when the thread enters a method, and another pointcut is reached when the thread exits the method.
4) Aspect: The combination of the pointcut and the advice is termed an aspect. In the example above, we add a logging aspect to our application by defining a pointcut and giving the correct advice.

In AspectJ we can use the pointcut-advice (PA) model 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
   }}
```

IV. ACCESS CONTROL AS AN EXAMPLE

The access control in java known as java access control (JAC) based on stack inspection works on the following three mechanisms

Basic permission checking: When a sensitive resource is about to be accessed, a call to the JAC API triggers a stack inspection algorithm, which checks whether all the classes in the current stack of execution possess the necessary permission to access the resource. If not, an exception is thrown. This basic behaviour prevents the confused deputy problem from happening, i.e., an untrusted class cannot lead a trusted one to access (or modify) a sensitive resource on its behalf.

Privileged execution: In some scenarios, it is necessary for a class to access a sensitive resource on behalf of another possibly untrusted—class. For this, the JAC architecture supports privileged execution.

Permission contexts: When accessing a sensitive resource, it can be necessary for a class to use the permissions present at another point in the execution of the application. The JAC architecture provides the means to capture a permission context and restore it later on.

While these three mechanisms together provide a very powerful access control system, the JAC architecture suffers from modularity issues. Indeed, in order to trigger permission checking through stack inspection, an explicit call to the JAC architecture is necessary. As a consequence, code
related to permission checking ends up scattered at each and every place where sensitive resources are accessed, tangled with other concerns. In other words, access control is a crosscutting concern, which pollutes and limits the good modularity of a system.

Consider a service whose function is to periodically clean a temporary directory. This kind of service could be useful, for instance, to applications that require a great amount of disk space or to disk-cleaning applications. Fig. 2 depicts the architecture of the service: TmpCleaner is the entry point for client applications through its clean method. Each file in the temporary directory is deleted by an auxiliary CleanTask instance, which encapsulates the deletion of a particular file. The service is designed this way because if the file cannot be immediately deleted, the CleanTask instance schedules itself for later execution with a certain delay using a preexistent java.util.Timer. When the delay is over, the Timer invokes the CleanTask.run method again (the asynchronous nature of this call is denoted by the double diagonal lines at the bottom of the figure). If the file still cannot be deleted, the CleanTask schedules itself again until the deletion can be finally performed.

.

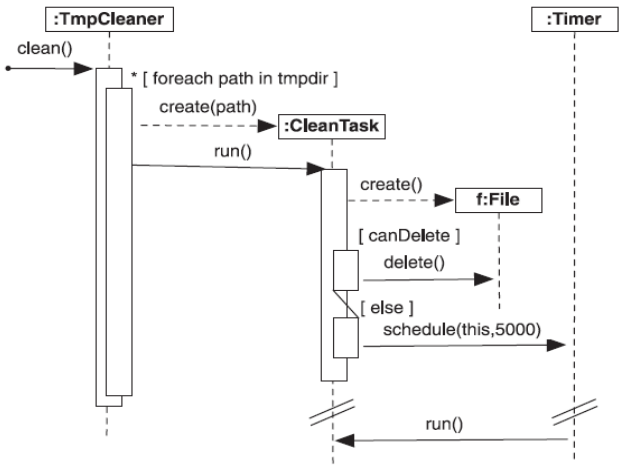


Fig.2. A service for cleaning a temporary directory.

A. Aspectizing basic permission checking

To determine whether a resource access is legal or not can be considered a matter of flow of execution: If the access is in the control flow of a class with a restriction to access the resource, then the access is illegal. Otherwise, the access is legal. Aspect languages usually provide means to reason about control flow: Aspects can be limited to seeing only the join points that occur within certain flows of execution. AspectJ, in particular, provides two options in this regard. First, AspectJ features percflow deployment, which creates and deploys a new aspect instance for each flow of execution starting in certain join points (specified using a pointcut). Second, AspectJ also features a cflow pointcut, which matches join points only under a particular flow of execution. A cflow pointcut can be conjuncted with other pointcuts as an extra condition over the join points to select. Using these AspectJ control flow features to limit the scope of restriction aspects, implementing basic permission checking is straightforward. Following is the FileDeletion- Restriction aspect, updated so that it only sees join points in the control flow of the execution of any method in UntrustedClient (additional restrictions are necessary for other untrusted entities). The code includes two specification options: using percflow (code labeled (a)) and using cflow (code labeled (b)). Only one is necessary.

```
Aspect FileDeletionRestriction
    Percflow(execution(*UntrustedClient.*()))   a
    Percflow(execution(*UntrustedClient.*()))   {
Pointcut resourceAccess(File file):
    execution(Boolean File.delete()) && target(file);
before (File file): resourceAccess(file) &&
    cflow(execution(* UntrustedClient.*()))   b   {
if(file.path.equals("……."))  {
throw new AccessControlException();
}
}}
```

*B. **Privileged execution***

The previous implementation of basic permission checking using either percflow or cflow is simple and direct. However, it cannot be used for realizing privileged execution. Once in a privileged execution, restriction aspects must not see a resource access even if it occurs in the control flow of a method of an untrusted class. In the case of percflow, once a restriction aspect has been deployed, it cannot be undeployed for the extent of a privileged execution, which is exactly what is needed. The case for cflow is similar.

It turns out that the scoping mechanisms offered by AspectJ do not suffice to express the JAC architecture stack inspection semantics. For this reason, it becomes necessary to manually manage state upon which to decide whether a resource access is legal or not. In the following, we detail a solution that maintains access control state through a pushdown automaton.

*C. **Access control automation***

In the following, we maintain access control state through a pushdown automaton. The novelty is that in this work we use an aspect-oriented approach for updating the automaton. With the use of this access control automaton, restriction aspects are deployed with global scope and must check the automaton to decide when to throw an exception. Each access control automaton10 has two logical states: LEGAL and ILLEGAL, representing legal and illegal resource accesses, respectively. For a given automaton, it is in state LEGAL when all classes in the stack of execution have permission to access the resource it guards. It is also in state LEGAL when a privileged execution (started by a trusted class) is in progress. If none of these conditions hold, the automaton is in state ILLEGAL. Fig. 3 depicts the access control automaton for privileged execution. This automaton is updated each time a (trusted or untrusted) class enters and exits the stack, and also when entering and exiting a privileged action. Transitions are prefixed with enter/exit and suffixed Trusted/Untrusted/DoPriv for this purpose. The stack of the access control automata is used to remember from which state an enter* transition originated so that the automaton can correctly switch back to the corresponding state on an exit* event. The symbols in the stack can be L, I, and *I, for "coming from LEGAL," "coming from ILLEGAL," and "coming from ILLEGAL but the last class was trusted," respectively. The *I symbol is useful to distinguish a privileged execution started by a trusted class and one started by an untrusted class. The _ symbol is used to ignore what is on the top of the stack (when specified in the second position of transitions), and to prevent a push operation from occurring (when specified in the third position of transitions).

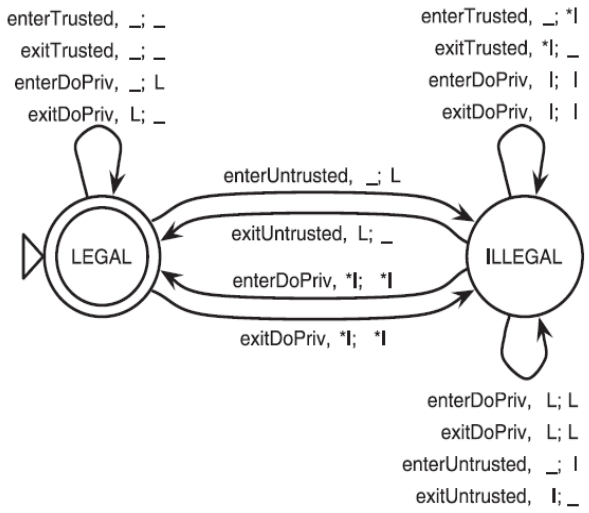
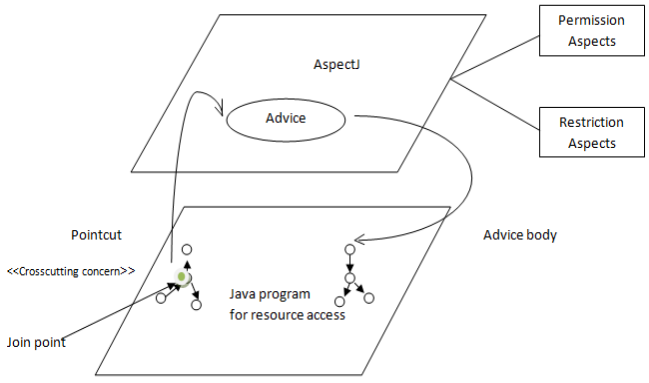


Fig.3. Pushdown automaton for privileged execution. Transitions have the form: <event>, <value on top of the stack>; <value pushed onto the stack>.

*D. **Permission contexts***

In order to support permission contexts, the access control automata can be extended with the necessary infrastructure for snapshotting and reinstalling a permission context. On the JAC side, a permission context is represented by the set of classes in the current thread of execution. This is so because the stack inspection algorithm needs the execution stack to state whether a resource access is legal or not. On the access control pushdown automaton side, the current states and stacks of the automata always instantaneously distinguish between a legal and an illegal resource access. Therefore, the permission context at an execution point can be considered as the set of access-control automaton instances at that point. In implementation terms, a permission context maps restriction aspects to automaton instances. The getContext method of the PermContext class can be used to get this map. It uses the getAutomaton method provided by each restriction aspect to get a copy of the current automaton instance.

ARCHITECTURE DIAGRAM



Deploying Permission aspect is equivalent to performing the explicit invocation to SecurityManager.checkPermission in File. Delete. However, the fundamental advantage of the aspect-oriented approach is that explicit calls to SecurityManager.checkPermission are no longer necessary.

Aspects like FileDeletionPermission simply delegate access control decisions to the default stack inspection algorithm. This algorithm ensures that all classes in the stack have the corresponding permission when a resource is about to be accessed.

```
FileDeletionPermission {
pointcut resourceAccess(File file):
execution(boolean File.delete()) && target(file);
```

```
before(File file): resourceAccess(file) {
SecurityManager.checkPermission(
new FilePermission(file.path, FILE_DELETE_ACTION)
);
}}
```

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

```
aspect FileDeletionRestriction {
pointcut resourceAccess(File file):
execution(boolean File.delete()) && target(file);
before(File file): resourceAccess(file) {
if(file.path.equals("…… ")) {
throw new AccessControlException();
}
}}
```

V. GENERALIZATION OF EXAMPLE

The deployment of each of the cross cutting concern described in previous section depends heavily on the actual type of the implementation of the particular application. In general it is not possible to define one set of aspects that will be applicable to all kinds of applications. A generic mechanism is needed that separates the implementation of security mechanism.

In the example described above you might have noticed that the deployment decisions are actually contained the pointcut definitions, i.e when and where an advice or aspect has to be applied. Also *cflow* and *precflow* constructs are used which match the join points under particular flows of execution. AspectJ has the ability to declare pointcuts abstracts and afterwards extend them to define the actual join points. Using this mechanism, it is possible to build a general permission or restriction aspect and redefine the included abstract pointcuts depending upon the specific application.

A mojor advantage of generalization is the ability to reuse the core structure of the security requirement. Since this will be similar for every situation, it is easy to extend and reuse the already available code. It should be properly designed once, after which aspect inheritance enables easy reuse.

```
Abstract aspect accessrequest  percflow (servicerequest ()) {
   Private Subject subject
   Abstract pointcut servicerequest();
   ….
   }

Abstract aspect accesscontrol  perthis (servicerequest ()) {
   Private Subject subject
   Abstract pointcut init(subject);
   ….
   }
```

REFERENCES

[1] L. Gong and G. Ellison, *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2003.

[2] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers, "*Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2*," Proc. USENIX Symp. Secure Systems, 1997.

[3] 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.

[4] B. de Win, B. Vanhaute, and B. de Decker, "*Security through Aspect-Oriented Programming*," Proc. Advances in Network and Distributed Systems Security, pp. 125-138, 2001.

[5] John Viega, J.T.Bloch, and Pravir Chandra "*Applying Aspect oriented programming to security*".

[6] Bart De Win, Joosen, and Frank Piessens "*Developing Secure Applications through Aspect oriented programming*".

[7] Bart De Win, Bart Vanhaute, and Bart De Decker "*How Aspect oriented programming can help to building secure software*?"

[8] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers "*Going beyond the Sandbox: An Overview of the New Security Architecture in Java™ Development Kit 1.2"*.

[9] Rodolfo Toledo, Angel Nu´n˜ ez, _ Eric Tanter "*Aspectizing java access control"*.