Aspect programming with a security example

Carlos Oviedo Álvarez, Eng.
Florida Atlantic University, Boca Raton, Florida USA, coviedo@fau.edu

Michael VanHilst, PhD.
Florida Atlantic University, Boca Raton, Florida USA, mike@cse.fau.edu

Abstract
New emerging developments to build robust software, such as generative programming, domain specific languages, meta-programming and so on are drastically modifying the way in which software is perceived and dealt with. Aspect programming is a new way of programming that allows the inclusion of cross-cutting concerns into system design and implementation not captured when building software systems under object oriented models. This paper gives a quick overview of *aspects* and aspect oriented programming and gives an example of how this new technique could be used to address security concerns. Because aspects are a valuable tool in building robust, secure and efficient software, it is worth exploring this innovative technology.

Keywords:
Aspects, security

Introduction

During the past 40 years, software engineering has greatly evolved. Proof of this evolution can be seen in the way in which the paradigms of software have developed into new concepts and techniques such as object oriented programming, UML, data-flow oriented design and software patterns. The well-accepted criterion after this process seems to have 3 main steps that should be followed in order to produce quality software: analysis + design, implementation and testing. Likewise, in today's software development, new practices and paradigms have emerged as a product of a new stage of evolution for software.

One of these new practices is aspect programming. The intention of this paper is to explore this new technique as a way to develop improved quality software and to make an overview of the contributions it may have to a very important field in software development as security. AspectJ, being the current main implementation for aspect programming, is taken as the principal source, for the development of the paper.

First, a quick historical review of when and where aspect oriented programming (AOP) was created is given followed by an explanation of the theoretical foundation for AOP. Finally, a quick overview of how it works in the inside is given followed by some conclusions.

Aspect Oriented Programming

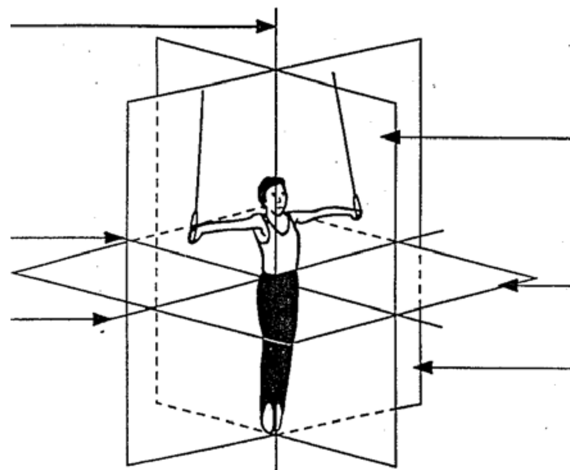
A little bit of history

During the late 90's the PARC (Palo Alto Research Center) [par05], an organization that has been at the forefront of new technological developments for more than 30 years, influencing major changes such as laser printing, the Ethernet and the graphical user interface, worked on forms of programming that make capturing complex design structures possible in a software implementation in a clean way. These forms included object oriented programming and meta-object modeling protocols.

Using their earlier work as a base, PARC focused on a way to capture what is called cross-cutting concerns or *aspects* of a particular system. Several years later we have AspectJ as a production system using Java as the base language to express cross-cutting concerns, administered by Eclipse, a non profit open platform for tool integration built by an open community of tool providers [ecl05]. The original new technology, under constant development by several commercial, academic and research groups [asp05c], has evolved extending several programming and description languages such as AspectC (C++) [asp05a], Aspect-oriented-PERL (Perl) [asp05d], AspectS (Smalltalk) [asp05e], Aspects (Phyton) [asp05f], AspectXML (XML) [asp04a], AspectC# (C#) [asp05b] and LOOM.NET (for the Microsoft .NET framework) [loo04], among others [asp05]; however AspectJ positions itself as the most complete framework of all.

Theoretical foundation

The central idea behind the aspect oriented programming (AOP) paradigm bases its methodology in the realization that there are issues or concerns that are not well captured by traditional programming methodologies [asp01]. These are called cross-cutting concerns.



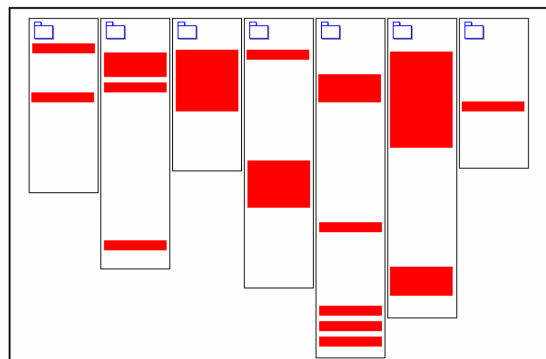
Cross-cutting [Cac04]

They describe concerns at the time of creating software solutions that can not be encapsulated in a given modular composition. The constructions provided by imperative object oriented programming languages (C#, C++, Java, Object Pascal, etc) [Cac04] were created to address the modeling of problems in a hierarchical view of the system and for these reason cannot capture cross cutting concerns correctly. These, may not need to relate to the core concerns directly, but are needed for a proper program execution.

They appear disseminated on the code, crossing different parts of the system not visibly related on its object-oriented model.

Georg, Reddy and France [Geo04a] conceptualize cross cutting concerns based on the fact that during any software development process, there are some requirements related to a particular concern that are spread across multiple requirements sources. This is indeed true for software concerns that are generally spread over the code of a system as for example logging (store an application's activity during a period of time), because we might want, in several execution points that are not necessary one after another in the program code, to "log" its activity in a text file or a database.

A consequence of cross cutting concerns in current systems is code spread across several modules. For the "logging" example, let us consider the next figure where all the classes in the system need some sort of logging for its actions. The sections in red represent parts of the code on each class where logging is taking place.



A specific concern spread along classes.

Security for instance, is a cross-cutting concern that every robust system should implement. Georg, Ray and France [Geo02a] argue that addressing a security concern during its design time requires taking into consideration the impact of the security concern on each design unit and modifying the affected design units accordingly. This can be a tedious and fault prone task that can result in an inconsistent and incomplete implementation of the concern. To address the issue, the methodology of aspect oriented programming can capture these kinds of concerns in a modular and maintainable way. In the next section AOP is discussed in more detail.

In general, cross cutting concerns can be identified if the same operation has to access several services in addition to its explicit function. Some other examples of cross cutting concerns are identity management, transaction integrity, authentication, and performance. [Lad02b] More work related how to encapsulate crosscutting concerns in software systems can be found in [Sch04]. In the next section, a very quick overview of aspects basic concepts gives a better notion of how these concerns can be mapped and modularized into code. It is based on the most complete implementation of a framework for aspects, AspectJ [asp01]

Aspects outline

Concerns such as security and logging, run across several units of modularity; in other words, across several classes in an object-oriented design. Because of this kind of entanglement, cross-cutting concerns are not reusable. They can not be refined or inherited. In summary, they are very hard to deal with. Object-oriented programming languages do not provide mechanisms to transform these concerns into classes.

Aspect oriented programming is a form of modularizing these cross-cutting concerns without breaking the underlying object oriented design. In AspectJ [asp01] and in general in AOP; a few new concepts are introduced: *join points*, *pointcuts*, *advices*, *inter-type declarations* and *aspects*. The Pointcuts and advices are going to affect the program's flow in a dynamic way; inter-type declarations on the other hand affect a program's class hierarchy in a static form and finally aspects, encapsulate these constructions. The dynamic parts of aspect programming correspond to pointcuts and advices, however to understand how they work it is vital to know what a join point is.

As was said previously *Aspects* are the unit to modularize cross-cutting concerns. In case of AspectJ [asp01], they behave like Java classes, but introduce pointcuts, advices and inter-type declarations.
A *join point* is identified as a specific execution point in the program's flow. A *pointcut* selects certain joint points and values at those points. An advice is the code (new code) that is going to be executed when a join point is reached. AOP also defines several *inter-type declarations* that allow modifying the program's static structure; that is, its classes and members as well as the relationships between its members.

Laddad [Lad02a] defines join points as specific points in a program's execution, a pointcut as the language construct that specifies join points, advices as the definition of pieces of an aspect implementation to be executed at pointcuts, and finally an aspect, as the structure that combines these primitives.

Consider for example a system whose purpose is the generation of figures [asp01]. A figure will be the conjunction of several figure elements, which can be points or lines.

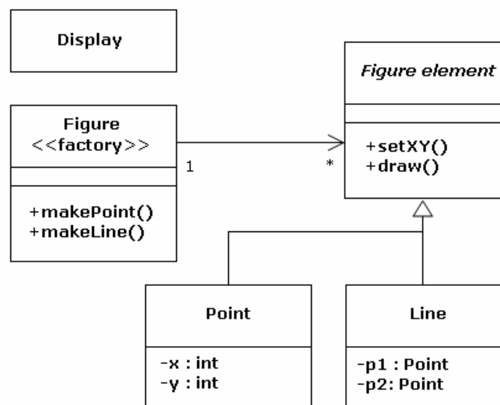


Figure example in UML

Pointcut

Under this example it is possible to identify several join points (execution points in the program). There are different kinds of join points, but for the sake of the example only one type is going to be considered, the most common one; a *call* join point.
Every *call* join point covers all the different actions of an object that is receiving a method call. That includes actions after starting evaluating all the arguments of the call up to when the method returns.

In AOP the pointcuts are going to pick out join points in the program code (flow), for example the next group of join points inside the "*move*" point cut are going to encompass the functionality related when movement of a figure.

```
pointcut move():

call(void FigureElement.setXY(int,int)) ||
call(void Point.setX(int))  ||
call(void Point.setY(int))  ||
call(void Line.setP1(Point))||
call(void Line.setP2(Point));
```

The dynamic capacity of join points and therefore point cuts; stands on the fact that every method call at runtime is an individual join point, and other join points can be running while a method call join point is executing.

Advice

To be able to implement the cross-cutting behaviors, *advices* are necessary. So far, point cuts can pick up several join points, but that is the only thing they do. Advices are the structures that allow some sort of functionality to execute (a body of code).
To understand its concept lets take a look of the advice examples:

```
before(): move() {
    System.out.println("about to move");
}

after() returning: move() {
    System.out.println("just successfully moved");
}
```

A before-advice runs as a join point is reached, and executes the defined code before the program proceeds with the join point; that is before the method starts running, after the arguments to the method call are evaluated. Conversely, an after-advice executes after the program proceeds with the joint point.

Inter-type declarations

Under AOP, not only the dynamic content of an existent program is likely to be modified but static (at compile-time). Inter-type declarations are declarations that cut across classes and their hierarchies [asp01]; going across multiple classes for example.

Summarizing, inter-type declarations are going to allow us to add static context to our program without having to modify the existent code. All the new desired capabilities are going to be local to a particular aspect's definition. The aspect then is going to be in charge of managing this new static "fields" for the existent classes we want to modify. This brings us to the definition of an aspect.

Aspects

An aspect is a wrapper. In other words, is the structure which encompasses pointcuts, advices and inter-type declarations in a modular way, very similar to "object oriented" classes. It could include methods, fields and initializers, in addition to the cross cutting members. Aspects are then a solution to modularize crosscutting concerns and its effects.

A typical aspect example is "logging"; on the next example we want to track when there is going to be a "move" on a figure (there is a point cut defined for that matter) and inform the user of that behavior.

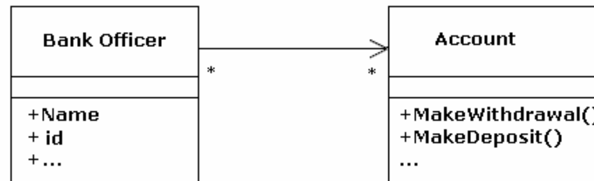
```
aspect Logging
{
    OutputStream logStream = System.err;

before(): move() {
        logStream.println("about to move");
    }
}
```

There are several ways to insert the cross cutting concerns code (aspects code) into existing programs; this process is called weaving [Cac04]. It can be achieved at a pre-compiling stage (the base code + aspects code are combined to produce new source code), at compilation time (base code + aspects code to produce the program's object code) and in a dynamic way controlling the program's execution, intercalating the aspects every time the code reach a join point included in a point cut for a desired functionality.

According to Cáceres [Cac04] two important reasons why aspects are starting to become so popular are its "obliviousness" and "quantification". The obliviousness refers to the fact that a programmer that is assigned to a specific task should not be aware of other dimensions that could affect his/her code. Quantification is the ability to indicate in which join points some aspect is going to be applied without naming them one after another.

Security example

A simple example of how aspects could be used for security could be to design an aspect to control access to a specific resource. In this case a Bank Officer of certain type is going to modify certain information of a particular customer's account in a bank.



Account access by a Bank Officer

We want to restrict access to the account object, enforcing authentication, and be sure that every time an action is performed over it some sort of checking could be done for the entity accessing it.
For this case the aspect that models this concern is defined as:

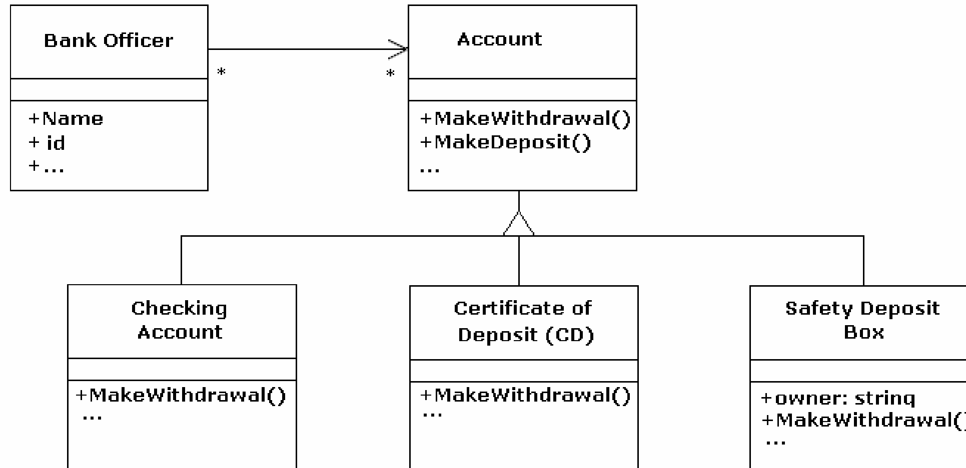
```
public aspect AccountAuthorization
{
    OutputStream logStream = System.err;

    boolean grantAccess(string id)
    {
      if(id != "guest")
          return true
      else return false;
    }

    Pointcut change():
        call(void Account.doWithDrawl());
```

```
        before(): change() {

            logStream.println("Change in progress...");
            if(!grantAccess(context.id))              throw              new
    UnauthorizedAccessException();
        }
    }
```

We would like to know, who is trying to make a withdrawal or a deposit.
For this case we would like to enforce the authorization capturing every time a bank officer wants to make a withdrawal from an account. The “AccountAuthorization” aspect is going to help us for this purpose.

We specified a pointcut named “change” that captures the join point defined by the “makeWithdrawal” method from the Account class. Every time this method is called, we are going to be aware of it. Since we are trying to identify who is trying to make a withdrawal before it is performed to determine if it has the necessary permissions or not, a “before” advice was defined, so that when a call to the “makeWithdrawal” method is intercepted then the “id ”from the active user in the current “context” will be checked. In case it does not have access to perform the operation, an exception will be thrown. The implementation of the “grantAccess” method is local to the aspect (for this simple example a person that is not a guest is allowed to do a withdrawal). All the reasoning about it can be done within the limits of the aspect.

Now, imagine a scenario where instead on having only one “Account” object, we would have a generic “Account” class and several subclasses associated with it. The next figure exemplifies this case creating 3 subclasses for the “Account” class: “Checking Account” “Certificate of Deposit”, and “Safety Deposit Box”.



Account class with 3 subclasses

The idea behind this is that we would have then 3 different types of “MakeWithdrawal” operations.
Normally, if we add this new functionality to a system we would have to add the authentication for each of the new sub-classes created; a fault prone and time consuming task. However, with the same aspect we defined for the first example, it is possible to manage the authentication for that operation in the new classes. The only change that would need to be performed in here is to change a little the line with the definition of the pointcut “change” to make it look like this:

```
Pointcut change():
        call(* MakeWithdrawl(..));
```

This will capture all the join points when any method with the name of "MakeWithdrawal" receiving any parameters is called. Since we are not mentioning specifically from which class is the method from, it will capture all of them. In other words, the point cut will enclose the join points for any call to any implementation with the name of "MakeWithdrawal", that is, any method available with that name.

Now, making this example a little more interesting, let's say we want to reuse the functionality written to perform authentication. At this point we count with an aspect that covers this functionality. Let's rename it as "SimpleAuthorization" and define it as an abstract aspect this way:

```
abstract aspect SimpleAuthorization{

    OutputStream logStream = System.err;

    public static boolean grantAccess(string id)
    {
      if(id != "guest")
          return true
      else return false;
    }

    abstract pointcut change():
        call( * Make*(..));

    before(): change() {

        logStream.println("Change in progress...");
        if(!grantAccess(context.id))
            throw new UnauthorizedAccessException();
    }
}
```

Having a similar behavior as an abstract class, it is possible to define an abstract aspect whose specialization can be written in other sub classes later. For the "SimpleAuthorization" aspect we will assume that all the methods that represent an action for our system will be named "Make" followed by the name of the action. Notice that the pointcut "change" will capture then the join point described by calls to any methods with the word "Make" in them.

To achieve the account authorization functionality from the first example it is just a matter of extending the aspect and specifying the classes that are going to be monitored for authorization of the current user on the system.

```
public aspect AccountAuthorization extends SimpleAuthorization{

    pointcut change(): within(Account) ||
                       within(CheckingAccount) ||
                       within(CertificateOfDeposit) ||
                       within(SafteyDepositBox);
    //...
}
```

The “within” keyword allows to specify the join points when the execution of the mentioned classes is accomplished. If we wanted to specialize this authorization for some other set of classes, a transaction for instance, we could do something like this:

```
public aspect TransactionAuthorization extends SimpleAuthorization {

    pointcut change(): within(Transaction) ||
                       within(SecureTransaction);
    //...
}
```

The “TransactionAuthorization” aspect will handle the authorization over transactions performed on a bank, as the generic class “Transaction” and the specialized subclass “SecureTransaction”.

Another use for aspects to enforce security, or any desired policy (concern), for the bank-account example; is to expose some of the pointcut context information to perform another type of checking over transactions on an account; for example verifying a large sudden withdrawal. The pointcut and advice that would envelop this functionality would be:

```
Pointcut change(int x):
        call(* MakeWithdrawal(int))
        && args(x);

before(int x) returning(x){
      logStream.println("Withdrawal in progress...");
      if(x > 100000) throw new WarningException();
}
```

What it means is that any a call to a method named “MakeWithdrawal” receiving an “int” value as a parameter will be captured by the pointcut “change”. Before it happens we want to check if the transaction exceeds a determined amount, to monitor suspicious activity. In our case a withdrawal for more than $100.000 will be monitored. Again, both the code from the account or its subclasses had to be modified, and the logic relating this concern was covered by one aspect definition.

This has been a quick overview of how *aspects* work. However a deeper and more detailed analysis is strongly suggested to gather a better perspective. More information on aspect programming and aspect design can be found at [asp01] [Lad02a] [Lad02b] [Geo02a]. One of the best sources for AOP related technology is [asp05c]

Conclusions and Related Work

After the present overview a quick historical review how aspect oriented programming appeared was performed followed by an explanation of the theoretical foundation for AOP as well as a review of its basic internal components; this, followed by an example for authorization using aspects. Based on that some conclusions that can be drawn:

Aspects are capable abstract structures that can capture cross cutting concerns such as security, and be applied to a system. They could be implemented on new systems as well on established systems; in this case a new compilation would be necessary.

Security concerns, when modeled using aspects can be maintained in one place (an actual piece of code that is applied in several parts of the code from a system), ergo, it is more likely to keep semantically correct than if it had to be inserted in a program with millions of lines in several places and then a new changed arrived.

Another example of how aspects might be used to enforce security could be to add functionality to record every time a class is created or destroyed, tracking who called a routine and recording when it was executed. This would enforce the principle of non-repudiation.

AOP seems to be a good approach to handle cross cutting concerns when building a system. At this point there are some ways for weaving (interlacing) aspects into a defined model using different technologies [Ras02a]. Georg and others [Geo02a] propose a technique for weaving aspects into a model by the use of role models to describe aspects, from the static and dynamic point of views. They mention a role model is a structure of roles where a role (meta-object) defines properties that must be fulfilled by conforming UML model elements. Using this representation as a design pattern it is possible to weave aspects into current static and dynamic parts of object oriented models (class diagrams, sequence and collaboration diagrams); this allows the design pattern to be reusable. Relating security concerns, design patterns are potentially reusable for different systems with similar characteristics.

At the moment, software development oriented to aspects is still on an early stage, under constant expansion, and it is worth exploring its characteristics and potential due its nature and ability to encapsulate cross cutting concerns, a vital feature when designing and implementing robust software, being security one of the most important.

References

[asp01] Introduction to AspectJ. http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/starting-aspectj.html

[asp05a] AspectC. http://www.aspectc.org/

[asp05b] AspectC#. http://www.dsg.cs.tcd.ie/index.php?category_id=169

[asp05c] Aspect Oriented Software Development. http://aosd.net/

[asp05d] Aspect-oriented-PERL. http://search.cpan.org/~marcel/Aspect-0.08/lib/Aspect/README.pod

[asp05e] AspectS (Smalltalk). http://www.prakinf.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/

[asp05f] Aspects (Phyton). http://www.logilab.org/projects/aspects/

[asp04a] AspectXML (XML). http://www.aspectxml.org/

[Cac04] Abdiel Cáceres González, AOP presentation. http://computacion.cs.cinvestav.mx/~acaceres/courses/udo/poo/

[ecl05] Aspect J, Eclipse Project. http://eclipse.org/aspectj/

[Geo02a] Geri Georg, Indrakshi Ray, and Robert France, “Using Aspects to Design a Secure System”, 8th IEEE International Conference on Engineering of Complex Computer Systems. December, 2002.

[Geo03a] Geri Georg, Robert France, and Indrakshi Ray, “Composing Aspect Models”, The 4th AOSD Modeling With UML Workshop, UML 2003, October, 2003.

[Geo04a] Geri Georg, Raghu Reddy, and Robert France, "Specifying Cross-Cutting Requirement Concerns", October 10-15, 2004

[Lad02a] Ramnivas Laddad “I want my AOP!, Part 1” http://www.cs.concordia.ca/~comp6431/material/reading-list/laddad02iwantmyaop-part1.pdf

[Lad02b] Ramnivas Laddad “I want my AOP!, Part 3” http://www.javaworld.com/javaworld/jw-04-2002/jw-0412-aspect3.html

[loo04] LOOM.NET (for the Microsoft .NET framework). http://www.rapier-loom.net/

[par05] PARC. Palo Alto Research Center. http://www.parc.com/

[Ras02a] Awais Rashid “Weaving Aspects in a Persistent Environment”, Computing Department, Lancaster University, Lancaster LA 1 4YR, UK

[Sch04] Christa Schwanninger, Egon Wuchner, Michael Kircher. “Encapsulating Crosscutting Concerns in System Software”. 2004

Biographic Information

Eng. Carlos OVIEDO ÁLVAREZ. Mr. Oviedo is a Masters student in the Department of Computer Science and Engineering at Florida Atlantic University. He graduated from the Technology Institute of Costa Rica in Computer Science. He received the LACCEI Scholarship Award, and CONICIT Scholarship Award from Costa Rica. He is an active member of the Secure Systems Research Group at Florida Atlantic University.

Dr. Michael VANHILST. Dr. VanHilst is an assistant professor at Florida Atlantic University. He received the BS and MCP degrees from MIT, and MS and PhD degrees from the University of Washington. He is a member of the Secure Systems Research Group at Florida Atlantic University. Dr. VanHilst worked in industry for many years before joining Florida Atlantic University.