



KUNGL
TEKNISKA
HÖGSKOLAN



HØGSKOLEN
I GJØVIK

NISlab

Norwegian Information
Security Laboratory

ASPECT-ORIENTED PROGRAMMING AND SECURITY

*A comparison between implementing
JAAS with AOP and OOP*

Hågen Hasle



Institutionen för
Data- och Systemvetenskap

Examensarbete
Nr 2004-x-164
2002

Examensarbete 20 poäng
i data- och systemvetenskap
inom magisterprogrammet i informations- och kommunikations säkerhet,
Kungl Tekniska Högskolan

ASPECT-ORIENTED PROGRAMMING AND SECURITY

A comparison between implementing JAAS with AOP and OOP

Hågen Hasle

Gjøvik University College, Box 191, 2802 Gjøvik, Norway

Abstract: Good programming is essential if we want secure software. Security flaws can occur in all phases of software's lifecycle, and proper design and sound configuration and environment are all important elements. But many security-holes are introduced by sloppy programmers and insecure programming. Aspect-Oriented Programming (AOP) is a new programming paradigm that among many things has been recognized as showing great promise when it comes to applying security. We have implemented a security framework into an existing application to measure how good AOP is to add authentication and authorization to existing applications. We define a set of metrics that can be used to test the different approaches and determine which one is most effective. Our results show that it is difficult to use standard OOP metrics to measure the advantage AOP gives you, even after adaptation. Our results still show that AOP show merit when it comes to making code reliable and maintainable.

Keywords: Information security, Aspect-Oriented Programming, JAAS, software engineering metrics.

På norsk: God programmering er essensielt hvis vi ønsker sikker programvare. Sikkerhetshull kan oppstå i alle fasene i et programs livssyklus, and god design og riktig konfigurering er viktige elementer for å unngå dette. Men mange sikkerhetshull introduseres av slurvete programmerere og usikker programmering. Aspekt-orientert programmering (AOP) er et forholdsvis nytt programmeringsparadigme som blant annet får anerkjennelse for at det virker lovende når det gjelder å sikre kode. Vi har implementert en sikkerhetsløsning både med AOP og vanlig objekt-orientert programmering (OOP) for å måle hvor bra AOP er til å legge autentisering og autorisering til en eksisterende applikasjon. Vi har definert et sett metrikker som kan brukes til å male forskjellene på de to fremgangsmåtene og avgjøre hvilken som er best. Resultatene våre viser at det er vanskelig å bruke vanlige OOP metrikker for å måle fordelene AOP gir. Likevel viser resultatene våre at AOP gir kode som er mer pålitelig og enklere å vedlikeholde.

TABLE OF CONTENTS

1.	Introduction.....	1
1.1	Description of key terms	1
1.2	Problem description	4
1.3	Claimed contributions	5
1.4	Justification and motivation	5
2.	Related work	6
3.	What we have done.....	10
3.1	Discussion about the case study	11
3.2	The programming.....	12
4.	Analysis of the differences between AOP and OOP	14
4.1	Code examples from the applications	14
4.2	Impact on the Trusted Computing Base.....	17
4.3	Hacking JAAS.....	19
4.4	Experiences from programming with AOP.....	20
5.	Quantifying and measuring the differences	21
5.1	Metric I: Lines of Code	23
5.2	Metric II: Code complexity	25
5.3	Metric III: Chidamber and Kemerer Metrics	30
6.	Conclusions.....	35
7.	Future work.....	36
8.	References.....	37

1. INTRODUCTION

The topics covered by this thesis are aspect-oriented programming (AOP) in Java and software metrics. More specifically how AOP relates to authentication and authorization as implemented in the Java Authentication and Authorization Service (JAAS) and how it is possible to quantify and measure the impact AOP has on security.

1.1 Description of key terms

In this chapter we will describe some of the key terms used in this paper. To be able to fully understand this paper a better understanding of these terms than what we can give here is probably necessary, but even a rudimentary understanding will help you a long way.

1.1.1 Java

Java is a high-level programming language originally made by Sun Microsystems. The following description is taken from webopedia.com:

Java is an object-oriented language similar to C++, but simplified to eliminate language features that cause common programming errors. Java source code files (files with a .java extension) are compiled into a format called bytecode (files with a .class extension), which can then be executed by a Java interpreter. Compiled Java code can run on most computers because Java interpreters and runtime environments, known as Java Virtual Machines (VMs), exist for most operating systems, including UNIX, the Macintosh OS, and Windows.

Java is the programming language we will focus on in this thesis. Although AOP can be used in several different programming languages, I have chosen to use Java because it is the programming language I am most familiar with. Java is also one of the largest and most widely used programming languages today, especially on the server side.

1.1.2 JAAS

The Java Authentication and Authorization Service (JAAS) is a package that enables services to authenticate and enforce access controls upon users. It implements a Java version of the standard Pluggable Authentication Module framework, and supports user-based authorization.

JAAS consists of two parts: *authentication* and *authorization*. In other words, it can be used for both authentication and authorization: For authenticating users to securely determine who is executing Java code and for authorization of users to make sure they have the right permissions required to perform their actions.

A typical usage of JAAS consists of four steps, from a programmers point of view:

1. Create a LoginContext
2. Optionally pass a CallbackHandler to the LoginContext, for gathering or processing authentication data
3. Perform authentication by calling the LoginContext's login() method
4. Perform privileged actions using the returned Subject (assuming login succeeds)

To successfully use JAAS, you also have to configure your application properly. JAAS requires several different files to be supplied.

1.1.3 Aspect-oriented programming

AOP was developed by researchers at Xerox Palo Alto Research Center, and was first introduced in [17]. The following are quotes from some articles about aspect-oriented programming:

Aspect-oriented programming (AOP) grew out of a recognition that typical programs often exhibit behavior that does not fit naturally into a single program module, or even several closely related program modules. Aspect pioneers termed this type of behavior crosscutting because it cut across the typical divisions of responsibility in a given programming model. In object-oriented programming, for instance, the natural unit of modularity is the class, and a crosscutting concern is a concern that spans multiple classes. Typical crosscutting concerns include logging, context-sensitive error handling, performance optimization, and design patterns. [1]

Object-oriented techniques for implementing such concerns result in systems that are invasive to implement, tough to understand, and difficult to evolve. The new aspect-oriented programming (AOP) methodology facilitates modularization of crosscutting concerns. [2]

AOP allows you to define crosscutting concerns that can be applied across separate, and very different, object models. It allows you to layer — rather than embed — functionality so that code is more readable and easier to maintain. [3]

AOP and OOP are not competing technologies, but actually complement each other quite nicely. [3] This means that AOP is an extension to OOP, and is not meant to replace it.

There are some new terms in AOP that are essential to understand and be aware of. We will describe them shortly here. For a more thorough description, please turn to [17] or [8].

Aspect-oriented programs consist of core classes and *aspects*. The core classes are regular OOP classes, and are meant to implement the functional requirements of the program. Aspects are separate modules that implement functionality that is orthogonal to the core functionality. This functionality is meant to be executed at locations in the core code that the programmer defines. These locations are called *joinpoints*. A *weaver* is responsible for insert the right code into the right locations. The code in an aspect is organized in *advice*. An advice is a construct that is similar to a method in OOP. An advice can be executed *before*, *after* or *around* a specified joinpoint. An around-advice can be used to replace the code in the core classes, or merely as a combination of a before- and after-advice. The last new construct in AOP is the *pointcut*, which is used to tie advice and joinpoints together. You use the pointcut to specify which joinpoints you wish to execute your advice at. A joinpoint can be all kinds of different code constructs, depending on the kind of AOP implementation you use. Examples are method calls, method execution, exception handling, attribute access and object initialization.

AOP can be implemented in several different ways in Java, which is the programming language we will focus on. [1] describes three ways; dynamically through Java's dynamic proxies-feature, dynamically through bytecode manipulation or statically through a kind of precompiler. The most famous AOP framework in Java, AspectJ, uses the last approach. Depending on the approach you use, you might need to use special tools or precompilers. AspectJ comes with a precompiler that turns the aspects into ordinary code, and weaves everything together. The compiled classes can be run on an ordinary Java Virtual Machine (JVM). If you use the bytecode manipulation approach, you will need a tool that manipulates the bytecode the ordinary compiler produces. Several such tools are made; AspectWerkz and JAC are two well-known options. You can still use an ordinary JVM. The last approach, dynamic weaving through Java's dynamic proxy feature, you will still need a framework that does the weaving, but it is all done in regular Java, no bytecode processing is necessary.

1.1.4 Common Criteria

Common Criteria [27] is a program for evaluation of IT Security. The Common Criteria represents the outcome of efforts to develop criteria for evaluation of IT security that are widely useful within the international

community. It is an alignment and development of a number of source criteria: the existing European, US and Canadian criteria (ITSEC, TCSEC and CTCPEC respectively).

The CC presents requirements for the IT security of a product or system under the distinct categories of functional requirements (CC Part 2) and assurance requirements (CC Part 3). The CC functional requirements define desired security behaviour. Assurance requirements are the basis for gaining confidence that the claimed security measures are effective and implemented correctly.

1.2 Problem description

In today's society security has become a prime concern for both people and companies. To achieve confidentiality and integrity for data we want to protect we need user authentication and authorization.

There are many applications already built without security or with security-solutions that doesn't meet the demands of the environments the applications are deployed in. An example of this is that instead of using a standard username-password combination as authentication there might be a need of a single sign-on mechanism or a link to the company's central LDAP-server. Because of this applications must be modified and recompiled. This can be a troublesome and error-prone process, and can possibly lead to new security-holes. It can also be quite time-consuming, due to the difficulty of the task.

To help programmers more easily build secure and reliable authentication and authorization-solutions frameworks like JAAS have been created. But security can still be troublesome to implement. Not only do we need to use the proper security mechanisms, but they must also be properly implemented.

JAAS-related code will still be sprinkled around at every place security is needed, possibly all through the application. This mixing of core functionality and security-related functionality puts great demand on developers; they need to handle both things at the same time. Not every programmer has the skills and competence required.

Aspect-oriented programming claims to help solve these problems with its new crosscutting techniques. Several papers [5, 6, 7, 9, 18] claim that AOP is a viable programming paradigm, especially for security concerns. AOP will lead to a better organized codebase, where security and core functionality is not mixed together.

A well structured and organized codebase is a good starting point if you want a secure application with few faults and errors. When an application evolves, authorization might need to be added to several new places in the code. It is important that the application is easy to maintain, so that bugs are not introduced in the process.

Common Criteria (CC) also recognizes this. CC's last part is about Assurance Requirements. An example of these requirements is design complexity minimization:

Design complexity minimisation contributes to the assurance that the code is understood -- the less complex the code in the TSF (Target of Evaluation Security Functions), the greater the likelihood that the design of the TSF is comprehensible. Design complexity minimisation is a key characteristic of a reference validation mechanism. [28]

AOP is quite new, and the claims made about AOP's superiority are not very substantiated. Our concern in this paper is thus how we can verify that AOP does indeed lead to code that is better organized, resulting in better reliability and maintainability.

1.3 Claimed contributions

Recently some papers have proposed some metrics that can be used to describe and measure the differences between AOP and OOP. We discuss this work, and also propose one metrics of our own for measuring growth in lines of code.

We show, through both a theoretical basis and a practical assessment using three different metrics, that implementing JAAS in AOP rather than OOP can lead to improvements in reliability and maintainability, which ultimately leads to improved security.

Our focus is not upon security functionality, because the same functionality can be developed with JAAS using both OOP and AOP. We focus instead on more traditional software metrics, measuring aspects like code complexity and dependencies. Although this is briefly explained above, a more thorough explanation of why we do this can be found in the chapter "Quantifying and measuring the differences".

1.4 Justification and motivation

If aspect-oriented programming can help to separate code that handles authentication and authorization from the main code of the application, ordinary developers won't need to be security-specialists. By finding a way

to modularize and reuse code for authentication and authorization we can hopefully reduce the complexity and trouble of retrofitting new security-solutions to existing applications. This modularization can ease the burden on developers and lead us to develop more secure applications faster.

Ordinary developers can focus on meeting the functional requirements and therefore specialize their competency, and companies can save money and still increase their software's security. AOP can also help companies and developers meet the demands set by the Common Criteria more easily.

2. RELATED WORK

A lot of papers on aspect-oriented programming and security agree that security is a very good fit for AOP.

In a quite old paper [13], the authors point out several interesting things:

- When adding AOP to existing code it is often necessary to refactor the existing code to produce good and viable join points.
- It is best when the interface between aspects and code is narrow and unidirectional. The aspect should have a well-defined effect on the base code and the aspect should refer to the base code but not vice versa.
- It is easier to understand and manage aspects when the aspect code forms the glue between two object-oriented structures.

The authors conclude that AOP shows promise and can benefit developers. But they also think we have much left to learn about AOP, and are unsure as to what kind of problems it is best suited for.

In [5] the authors write that security is a good target for AOP. Security often crosscuts very deeply with the application. Also, the security implementation needs to be implemented very carefully in order not to introduce security holes. Verification of correctness is much easier if the code is not scattered across many classes in the application. In [6] the authors argue that AOP shows great promise as a technique for building more reliable software and sees security in general as an area where AOP could greatly ease the burden for developers. In [7] the authors write that AOP can help us separate security-related concerns from the main code and thus freeing developers that are not security-experts of having to think about security all the time.

Several papers show that AOP is usable for adding authentication and authorization to applications. [9, 18] shows examples of how this can be

done. One of the few books on AOP [8] has included a chapter about how AOP can be used to write JAAS-code that implements authentication and authorization. In [10] the authors describe how infrastructural services can be added to objects with AspectJ. They show both authentication and authorization, but also persistence and transactions.

Other papers also point out security in general and more specifically authentication and authorization as crosscutting concerns that are well suited to be modularized as aspects. But we have not found any papers with an emphasis on adding authentication and authorization to *already existing* applications. A question still remains if AOP is more or less useful if the developer isn't free to structure the code the way he wants, but has to adjust his new code to an already existing codebase.

When it comes to adding authentication and authorization with AOP, there are several ways this can be done. In [1], three ways to implement AOP is described, all with their pros and cons. Static weaving requires recompilation of code, but doesn't necessarily require great modification of the existing code. Dynamic weaving is a form of byte-code manipulation, as doesn't require us to recompile the code. Dynamic proxies require both a recompile and more extensive modification of the source code.

In [5] the authors describe a security framework they are building in an aspect-oriented way using AspectJ. AspectJ uses a static form of weaving. They have learned by doing this that AspectJ is not flexible enough, it needs a more flexible, open weaving process. The authors of [10] also conclude that AspectJ is not well suited for creating generic frameworks. This is not because of the static weaving, but because the aspect's code and the declaration of where the aspect should be used is not separate. We think this work has been done with an older version of AspectJ, and the problems pointed out in these papers are solved in the newest versions. Our experience is that AspectJ is well suited for creating generic frameworks. It has the possibility to weave both sourcecode and compiled bytecode, both classfiles and jarfiles. And you can define abstract aspects with abstract pointcuts that lets you separate the aspect's code and the declaration of where it should be used. This shows that AOP is a new and evolving technology, where demands and wishes from the community are listened to, and where problems are worked out.

Some papers points out that the fact that AOP promotes a separation of concerns makes modularization easy, and this makes reuse easy. According to the authors of [14], the advantages of AOP are the following:

- Evolvable security concern enables agile development and maintenance. Instead of the impossible “get-it-right-the-first-time” it is possible to have a controlled “penetrate-and-patch” approach. Upgrading and evolving the application becomes cleaner, easier and more powerful.
- Better comprehensibility and focused efforts lead to fewer bugs. By modularizing security the security experts can focus on what they do best, and the regular developers can focus on implementing the functional requirements and worry less about security. It also helps reuse and parallel development.

AOP is not without problems though. According to [11, 12], there is a risk that a dynamic AOP framework can be used by an intruder to introduce malicious code into an application.

Many of the papers we have read have provided few measures and real experiments as proof of the claims they make. Many authors seem to base their claims of AOP’s superiority solely on their own experience. This is hardly scientific, and further research is definitely needed.

Until recently, there were few exceptions to this. [13] was one of the few exceptions. They have conducted a number of studies to assess the usefulness of AOP and similar technologies. Both semi-controlled experiments and larger case studies have been performed.

[12] also measures performance. Their metric is based on the execution speed and throughput.

In their re-engineering attempt of an existing application, the authors of [21] shows that using aspect-oriented techniques reduce the amount of lines of code in the application.

Quite recently, a new set of papers have appeared, where OOP and AOP is compared using traditional software metrics. Our work builds upon these papers and the work done by their authors. But before we describe them, we should first mention some important papers in the field of software metrics. Most of the work we present here is also described by Fenton in his book about software metrics [20]. His comments and critique has been valuable to us in finding the metrics we wanted to use, and in our work in general.

One of the most important papers about software metrics is [4], where Chidamber and Kemerer propose a suite of metrics that can be used for measuring and evaluating object-oriented code. They propose six metrics:

- Weighted Methods Per Class (WMC)
- Depth of Inheritance Tree (DIT)
- Number of Children (NOC)
- Coupling between objects (CBO)

- Response For a Class (RFC)
- Lack of Cohesion in Methods (LCOM)

These six metrics have since their introduction been greatly used and discussed, and form the basis of other metrics. They have also recently been adapted to be used for measuring AOP code.

We describe the Chidamber and Kemerer metrics suite (C&K metrics) in more detail later in this paper, when we use them to measure our own work.

Another important software metric is McCabe's cyclomatic complexity. [24] We describe this in more detail in the metrics chapter, but the essence is that we can measure the number of linearly independent paths through a module, and this is used to represent program complexity.

McCabe has also proposed a second metric, the essential complexity measure. The purpose of this metric is to express the level of structure in an application. There are some elements of a flowgraph that are considered structured. If a piece of code is expressed as a flowgraph we can remove those elements. This is called decomposing the graph. The essential complexity is basically the cyclomatic complexity of what is left of the flowgraph, which represents the unstructured code. The GOTO-statement is a well known source of unstructured code. Java is a high-level language where the GOTO-statement doesn't exist; it has only the constructs McCabe refer to as a structured prime. The essential complexity measure doesn't make much sense when measuring Java code, as all code according to McCabe's definitions is well structured.

Halstead's software science measures were an early attempt to measure program size and complexity. He counts the number of operators and operands, and defines among others a program's volume and level based on computations with these variables. Fenton criticizes Halstead's metrics in his book [20]. He argues that the relationship between the real world and the mathematical model is unclear, and that some of the metrics Halstead define are unvalidated and not properly articulated.

Bache introduced the VINAP measures in 1990, which measures control flow complexity. These metrics have been testing in several empirical studies, and are shown to be reliable indicators of maintainability. However, they are not as widespread as McCabe's cyclomatic complexity measure. They are implemented in two commercial measurement tools, but we have not found them in any open source tools. Papers describing them are also hard to find.

The last three metrics are all described in [20], and further references can be found there.

Aspect-oriented programming has been around for some years now, but compared to both procedural programming and object-oriented programming, it is quite new. This is reflected also in the research done in the field of software metrics. As mentioned above, papers about metrics for aspect-oriented programming and design have just recently begun to appear.

Zakaria and Hosny [18] discuss the effect AOP has on the Chidamber and Kemerer metrics (C&K metrics). Their work is mostly theoretical, and based on their thoughts and a few code examples they describe how the different metrics will be affected in a positive or negative way by using AOP.

Tsang, Clarke and Baniassad [19] have done almost the same as we do in their research. They have used the C&K metrics to compare an AO system with an OO counterpart. In doing so they had to adjust the C&K metrics to be able to measure AO code properly. We discuss their work and the adjustments they had to do to the C&K metrics in the metrics chapter.

Coupling is one of the metrics that Chidamber and Kemerer introduced. This metric has been important to other researchers as well, and a lot of papers and tools focus on it. Coupling is closely related to measuring dependencies. Zhao [22] proposes a metric suite for measuring the dependencies in AO code. He defines these metrics based on a dependence model which consists of a group of dependence graphs. The metrics he proposes can be used to measure the complexity of the code.

Dufour et al. [23] examine the dynamic behavior of AspectJ. They present a benchmark and a tool that can be used to measure the performance of AspectJ. Performance can be an important factor, also when it comes to security. The many DOS attacks on large web sites have shown that. We have chosen not to investigate this any further in our thesis. The performance of the different AOP tools is constantly increasing, and we feel that measurements done in this area will soon lose their importance.

3. WHAT WE HAVE DONE

We have added JAAS authentication and authorization to the Java Pet Store, an example application from Sun's Blueprints program. We have

developed two parallel and equivalent versions of this application, one using OOP and another using AOP.

By developing the same framework in two different ways we have been able to perform a case study. We have defined the variable we have control over as the programming paradigm. By changing this from OOP to AOP we can measure how other dependent response variables change.

We believe this kind of case study/model-building is appropriate for this task because the only way to test a new programming paradigm is to program or to view the programs of others. Others have chosen the same approach [19]. We discuss this further below.

3.1 Discussion about the case study

A formal experiment on requires appropriate levels of replication (several teams and/or several projects), and randomized experimental subjects and objects. A case study is when you examine objects across a single team and a single project, or when you don't have the required level of random selection. When doing a case study, it is possible to show the effects of a technology in a typical situation, but not generalize it to every possible situation. The advantage of an experiment is that it is easier to generalize your results. A case study has several elements that are similar to those of a formal experiment though.

We have defined an experimental hypothesis H1 stating that AOP will lead to better reliability and maintainability. We will test this hypothesis against our conservative hypothesis H0 which states that there are no significant differences. We will program the same application twice in two different ways, and this kind of comparative analysis is essential in experiments and case studies. When we program the authentication and authorization in a regular object-oriented way, this is the control, the status quo. The metrics we have defined are the response variables, which are used to evaluate the hypothesis.

A case study is a comparative study, and we have chosen the "sister project" variant. This kind of study involves two projects, and we implement the same framework twice. It is important that the state variables are similar for both projects. The state variables are the descriptive variables. If they're not constant it is more difficult to explain a change in the response variables.

Another key issue to be able to properly establish and measure the response variables is to minimize the effect of confounding factors. Confounding factors are when the effect of several factors is indistinguishable from each other. An example of this is when you're learning how to use a method or tool as you try to assess its benefits. We have tried to avoid this by spending a considerable amount of time learning AOP before we begun programming. The code we have written is influenced by what other, more experienced programmers have done before us. We have also let other programmers with expertise in this field review our code, so that possible improvements could be spotted.

One might argue that a formal experiment would be preferable to a case study. Can we generalize our results at all, or will the best we can say be that AOP works better than pure OOP *for us*? This is a possible weakness, but we don't see it as a viable option to let other people do the programming for us. To be able to do that, they must spend a lot of time learning JAAS and AOP. AOP is still so new that very few people have any experience with it. Our solution to this problem, besides the things we mention above, has been to make it easy for others to reproduce what we have done, and possibly do it better. We use publicly available applications and frameworks like the Java Pet Store Application and JAAS, so anyone can do what we have done. If anyone can improve either of the security frameworks it will be possible to produce a new set of results and compare them to our own. We believe this can help justify the fact that we have done the programming ourselves instead of letting others do it.

The description given here of a formal experiment and a case study is mostly taken from [16], and we recommend this paper for a more thorough explanation.

3.2 The programming

The Java Pet Store is an application made to teach new developers design patterns and good programming practice, and can be downloaded from <http://java.sun.com/developer/releases/petstore/>.

We have focused on the admin-application in Java Pet Store. This is a small application used to administer the store. It is a Swing application, and is normally downloaded from the web and started through Java Web Start (<http://java.sun.com/products/javawebstart/>). The admin-application already had a rudimentary form of security. Because Java Web Start is used to start the application, the entry point is a web page. This page has a form-based

authentication scheme, where the user is checked against a database of valid usernames and passwords. After this point, the only authorization that is made is on the servlet that process requests from the Web Start client, and the authorization is simply a check to see if it exists a session. (This implies that a user has been authenticated through the form-based authentication scheme.) This is a home-grown solution that works well enough, but it has certain flaws. It is very simple, and lacks any differentiation among the various authenticated users. It doesn't follow any standards, and if you want to check the users against a LDAP server instead of a database, a lot of programming must be done.

But the shortcomings of the authentication and authorization scheme used in the admin-application in Java Pet Store are not really our focus. Our focus is to add JAAS to an existing application, and we have chosen Java Pet Store because it is readily available and well known. So we started by removing the existing authentication. This is easily done in the web.xml file of the admin-application. Then we removed the authorization in the servlet. To do this we only had to alter one variable.

Because an application started by Java Web Start is a normal desktop application, and not an applet or another kind of special web-application, we could now start the admin-application from a DOS-prompt. We now had an excellent starting point for our own task; to add JAAS to the admin-application with ordinary object-oriented programming and with aspect-oriented programming. We needed to do four things:

We started by adding authentication on the client. The authentication we use is a simple form where the process use Windows' startup authentication and return the username and id the user has in Windows. This way no interaction with the user is necessary.

We then added authorization in the client in two ways. We check when the users select the About-item in the menu. Although the about action is a trivial one, it can still be used by us to show how adding authorization must be done. We also limit the users GUI. There are two different panels in the admin-application, the orders-panel and the sales-panel. We limit the access to the sales-panel by removing the button and menu-item that opens it if the user doesn't have the right permissions.

We then transfer the authenticated subject from the client to the server. The admin-application is a client-server application, and authorization must also be done at the server.

At last, we added authorization to the three actions the user can perform on the server.

We did all of this using both OOP and AOP. Both of the applications we developed can be reviewed, and can be found in a zip-file accompanying this paper.

4. ANALYSIS OF THE DIFFERENCES BETWEEN AOP AND OOP

In this chapter we will analyze and discuss some of the differences between AOP and OOP. This chapter serves as an introduction to the next chapter, where we focus on what we can measure and assess. We start of by presenting some code examples from the applications we developed, to show how the two programming styles differ. Then we discuss the impact this has on the Trusted Computing Base of an application. Last, we analyze how AOP and JAAS can make hacking an application possible, and the dangers we must be aware of.

4.1 Code examples from the applications

The differences between AOP and OOP can easily be shown in our source code. The class `ApplRequestProcessor` in the package `com.sun.j2ee.blueprints.admin.web` can serve as an example.

First, for every request we send the authenticated subject from the client to the server, appended to the original request. On the server-side, the subject must be read from the request and stored. In the OOP version, this is done in the following way:

```
ObjectInputStream oin = new
    ObjectInputStream(req.getInputStream());
try {
    sub = (Subject) oin.readObject();
} catch (Exception cnfe) {
    out.println(replyHeader + "<Message>" + cnfe.toString()
        + "</Message>\n" + "</Response>\n");
    return;
}
BufferedReader inp = new BufferedReader(
    new InputStreamReader(oin));
//req.getReader();
StringBuffer strbuf = new StringBuffer("");
```

In the original version, the code looked like this:

Aspect-oriented programming and security

```
BufferedReader inp = req.getReader();
StringBuffer strbuf = new StringBuffer("");
```

Then, when the original request is processed, one of three actions is executed. The actions are all wrapped by a `Subject.doAsPrivileged`-method, like this:

```
if (reqType.equals("GETORDERS")) {
    return (String) Subject.doAsPrivileged(sub,
        new PrivilegedExceptionAction() {
            public Object run() throws Exception {
                return getOrders(root);
            }
        }, null);
}
```

Inside the actions (in this case the `getOrders`-method), the users permissions are checked:

```
String getOrders(Element root) {
    AccessController.checkPermission(new
        PetStoreActionPermission(
            "GetOrdersAction"));
}
```

These changes are not big, but they are done at three different places in the file. Someone wanting to understand how authentication and authorization is added to the application will have to search every file to find out.

The AOP version is quite different. Here, the original code is unchanged, and instead we have added some advice and pointcuts.

First, let us look at the pointcut and advice that reads the authenticated subject from the request. We can see that the actual code is very similar to the code in the OOP version. This is natural, considering that the two versions do the same thing. The difference lies in the AOP version's pointcut expression, and how the code is organized in a separate method (advice) instead of integrated into the original code.

```
public pointcut doPostOperation(ServletRequest req )
    : call( BufferedReader ServletRequest.getReader() )
    && within( ApplRequestProcessor )
    && target( req );
```

```

Object around(ServletRequest req)
    : doPostOperation( req )
    {
        try {
            ObjectInputStream oin = new
ObjectInputStream(req.getInputStream());
            authenticatedSubject = (Subject) oin.readObject();
            ((HttpServletRequest)
thisJoinPoint.getThis()).log(authenticatedSubject
                .toString());
            BufferedReader inp = new BufferedReader(new
InputStreamReader(oin));
            return inp;
        } catch (ClassNotFoundException cnfe) {
            ((HttpServletRequest)
thisJoinPoint.getThis()).log("Exception:", cnfe);
        } catch (IOException ioe) {
            ((HttpServletRequest)
thisJoinPoint.getThis()).log("Exception:", ioe);
        }
        return proceed(req);
    }

```

Next, let us look at the code for the authorization checks. The pointcut is very simple here, one line is sufficient. There are two advice, this is similar to how we implemented the OOP version, where we had to insert code at two different locations.

```

public pointcut authorizationOperations()
    : call( * ApplRequestProcessor.getOrders(..) );

Object around()
    : authorizationOperations()
    && !cflowbelow( authorizationOperations() )
    {
        try {
            return Subject.doAsPrivileged(authenticatedSubject,
                new PrivilegedExceptionAction() {
                    public Object run() throws Exception {
                        return proceed();
                    }
                }
            );
        }
    }

```

```
        }, null);
    } catch (PrivilegedActionException pae) {
        throw new AuthorizationException(pae.getException());
    }
}

before() : authorizationOperations()
{
    AccessController

    .checkPermission(getPermission(thisJoinPointStaticPart));
}
```

We see that AOP has enabled us to group all the authorization code in one class, removing the scattering of code that we saw in the OOP version.

4.2 Impact on the Trusted Computing Base

A Trusted Computing Base (TCB) is defined as the following:

- A component or set of components that implement the security policy for a system. (<http://www.sendo.com/kb/glossary.aspx?ID=441>)
or:
- TCB refers to the totality of protection mechanisms (hardware, firmware and software) that provide a secure computing environment. The TCB includes everything that must be trusted -- access control, authorization and authentication procedures, cryptography, firewalls, virus protection, data backup, and even human administration -- in order for the right level of security to work. (<http://www.webopedia.com/TERM/T/TCB.html>)

Common Criteria uses a different terminology. A system being evaluated in Common Criteria is called a Target Of Evaluation (TOE). Their concept TOE Security Functions (TSF) is similar to what we refer to as the TCB. In [28], TSF is defined as “All parts of the TOE which have to be relied upon for enforcement of the TSP.” TSP stands for TOE Security Policy, and are the rules defining the required security behavior of a TOE. We see that this definition is quite similar to our first definition of TCB, a TSF is the implementation of the security policy.

In our case study, we only focus on the part of the TCB that lies within the software. Firewalls, virus protection, data backup and human administration is outside our scope. We focus primarily on access control,

authorization and authentication. The important part of these two definitions is that the TCB refers to the protection mechanisms, the components in the system that implement the security policy. In our application the protection mechanisms are our usage of JAAS.

We see from the code examples in the previous chapter that AOP permits the JAAS-related code to be kept separately from the core code. In our AOP version of the Java Pet Store, not a single line of code in the original application had to be changed to include JAAS. This means that all code that is a part of the protection mechanisms is separated from the core code. The aspects we have made are the components that constitute the TCB.

This isolation of the TCB makes it easier to understand it, and see how it is implemented. Tasks such as code reviews are simplified. This is relevant not only for day-to-day development and testing, but also for larger-scale projects where external evaluation is done.

An example of such an external evaluation scheme is the Common Criteria. In CC's part about Assurance Requirements, there are several requirements where the modularization provided by AOP is relevant and can be of big help. The following are quotes from [28], where CC's requirements are presented:

The ADV_IMP.2.2E element defines a requirement that the evaluator determine that the implementation representation is an accurate and complete instantiation of the TOE security functional requirements.

(page 99)

To be able to determine if the implementation of the TCB satisfies the security policy, it is a clear advantage that the TCB is organized in a few classes and packages, instead of being scattered throughout the sourcecode.

Other CC requirements are also relevant to our work, among others the requirements regarding Modularity, Reduction of complexity and Minimisation of complexity (ADV_INT), and the requirements on Representation correspondence (ADV_RCR).

It is however important to distinguish between the protection mechanisms and the code that they protect. Although AOP groups all the code related to the protection mechanisms together in one module, it also makes the link between the protection mechanisms and the code it protects more difficult to see. The declarative weaving-process AOP uses is not as easily understood as the ordinary OOP syntax where the JAAS code is put where authorization is wanted. So although it is easier to verify that the

protection mechanisms are properly coded and don't contain any bugs or backdoors, it is more difficult to check whether every part of the code you want to protect really is protected, nothing more and nothing less. (A pointcut can possibly select more joinpoints than you want as well as fewer.)

To help remedy this potential problem, AspectJ comes with an aspect browser, that lets you see which parts of your code a pointcut affects. This is a big help in designing the pointcuts properly, and is a very important tool.

4.3 Hacking JAAS

As we mention in the chapter on related work, a couple of papers discuss the possible dangers of using dynamic AOP. We will use this chapter to discuss this a bit further, but would first like to show how easy it is to disable every authorization check in a system implementing JAAS.

As written by J. Viega and G. McGraw in [15]; sometimes you have to demonstrate to people the flaws in their software, otherwise they won't believe you.

To mangle this treat, all we had to do was write these few lines of code:

```
public pointcut hackJAAS();
    : call( * AccessController.checkPermission(..) );

void around() : hackJAAS()
{
    //Do nothing. No proceed-call.
}
```

The reason this is such an easy task is that JAAS is a standardized framework. To perform an authorization check, you must call `AccessController.checkPermission`. Everyone knows this, both lawful programmers and hackers. That means that if you know that an application uses JAAS, you automatically know which code you need to disable. You don't need to see the sourcecode, you don't need to see any kind of documentation.

However, nothing is that simple. You still need to be able to weave the code we have written into the application you want access to. This will be the tricky part. As we explained in the chapter on key terms, there are several kinds of AOP. We distinguish between static weaving and dynamic weaving. Static weaving is done before the compilation takes place, or as a

part of the compilation. The aspects and the core classes are merged together into a common bytecode. This is the safest choice, and you only need to be certain that nobody tries to sneak any malicious code into the compilation process.

Dynamic weaving is a kind of bytecode manipulation, and this weaving takes place at run-time. That means that malicious code can be introduced when the application is started. This makes it easier for an insider to sneak code into the application, and the implication is that you not only need to be aware of malicious code when you compile the application, but every time you start it.

4.4 Experiences from programming with AOP

Adding authentication and authorization using AspectJ turned out to be quite an easy task, although we did run into some problems. This may be explained by the fact that AOP is a new programming paradigm for us, and learning to master something new is always a bit time-consuming. We used “AspectJ in Action” [8] as a basis for what we did. We ran into a problem that had to do with the classloading-mechanism in Java and permissions. It seemed that AspectJ changed the default classloader, and when the code used the classloader to load images, it didn’t have the proper permissions to load the images anymore. When we consulted the AspectJ usergroup about this, Ramnivas Laddad was so interested he wanted our entire codebase, so he could examine this behavior more closely. AspectJ is not supposed to affect the core code, but in this situation it did.

Using AspectJ was quite easy and very elegant, at least as long as we only wanted to insert authorization checks at certain easily identifiable join points. When we wanted to change how the GUI was displayed, it became slightly worse. The code that adds buttons and menu items uses the same class several times, so it was more difficult for us to construct a pointcut that only matched one joinpoint and not the other. One version of our code had code in the advice that checked the button or menu item to see if it was the right one. This worked, but was not an ideal solution. We had to have two different advice, one for the buttons and one for the menu items. And we had code in the advices that really belonged in the pointcut. Because of this reuse and proper abstraction/modularization was not possible, and this was clearly a bad design. Eventually we managed to create a better solution. We could reuse the same advice for both the menu items and the buttons, and we moved the code that singled out the right object into the pointcut. AspectJ is a powerful language, and the possibility to use if-statements in the pointcuts saved us.

All in all, programming with AOP and AspectJ was both powerful and elegant, but also somewhat demanding. As with every other tool or programming language, you have to expect a learning period where you make mistakes and encounter problems. Our experience is that programming with AOP also requires you to have a more thorough understanding of the libraries you are using. Understanding how JAAS worked and how we should implement it was easier in the OOP version. As AOP can lead to a better separation of programming responsibilities, the programmers who implement the authentication and authorization will more likely be experienced in JAAS, so this should not be a problem.

We didn't have to change the code at all when we used AOP, and the solution we ended up with seemed very elegant. This is of course a subjective opinion, and elegance is in the eye of the beholder. How does the two solutions compare when we try to measure and quantify the differences?

5. QUANTIFYING AND MEASURING THE DIFFERENCES

As we stated in the introduction, what we are trying to measure here is whether or not AOP can help to separate code that handles authentication and authorization from the main code of the application. Thus the metrics we use are not security metrics, but metrics measuring code complexity and similar aspects. This may sound strange, considering the fact that it is, ultimately, security we are interested in. We will explain the reasoning behind this now.

Security has many facets. One facet is the features implemented in a piece of software. If you make a web-application, you might want to use SSL to protect important information the user enters. If you don't, malicious attackers can possibly eavesdrop on the communication between the browser and the server and collect the sensitive information. Another facet of security is how well a feature is implemented. Bugs in an application can be dangerous. In C, buffer overflow attacks are well known. There are several examples of how an implementation of a security feature such as SSL has either bugs or logical flaws that are possible to exploit. If the web-application in the previous example uses a faulty SSL implementation, a malicious attacker can still get access to the information he wants. The moral is that not only do we need to use the proper security mechanisms, but they must also be properly implemented.

These two security facets correspond to the two parts of the Common Criteria specification, the functional requirements and the assurance requirements.

JAAS is one of the security mechanisms we can use in a Java application. It is a standardized solution to a common problem, how to implement authentication and authorization. Several of the components needed in JAAS are implemented by Sun or various third-party companies. These components have probably gone through extensive testing, and will for the sake of this argument be considered trustworthy. Our focus is on the code we write ourselves.

What we have done is to implement a secure solution based on JAAS, and added it to the Java Pet Store. The two versions we have made are functionally equivalent, although they are programmatically quite different. It is therefore irrelevant to measure anything related to the first facet of security that we mentioned above. What we have chosen to do is to measure attributes related to the second facet we mentioned. A well structured and organized codebase is a good starting point if you want a secure application with few faults and errors. The metrics we have chosen measure these attributes.

The two most important attributes we are interested in are reliability and maintainability. Reliability is closely related to security, and needs no further introduction. Maintainability is also important. When an application evolves, authorization might need to be added to several new places in the code. It is important that the application is easy to maintain, so that bugs aren't introduced in the process. It is easier to perform a code-review on a well structured and organized codebase than a complex and unstructured one.

As Fenton [20] describes in his book, we distinguish between internal and external attributes when we measure a product. Internal attributes includes code structure, function points and lines of code. These attributes can be measured before the application is finished. They can often be measured as early in the process as the design phase, or at least in the implementation phase, when you have access to the sourcecode. External attributes includes reliability, security, usability and maintainability. Most of these attributes are impossible to measure before the application is finished and put in production.

Reliability is a good example. To be able to measure the downtime of the application, or the mean time between failures, it is obviously necessary to be in the production phase. However, we would like to be able to predict the reliability of the software we develop before we get to the production

phase. To do this, we must find a correlation between the internal attributes we can measure, and the external attributes we are interested in.

There are several studies that have found such a correlation. Many studies are described in [25], and they show that McCabe's complexity metrics correlate with reliability.

As described in Fenton, a study by Li and Henry that used both the C&K metrics and McCabe's metrics (among others), claimed that the C&K metrics "contribute to the prediction of maintenance effort over and beyond what can be predicted using size metrics alone."

We have used three different metrics to compare the two solutions we developed. They are described in this chapter, along with the results we got when we applied them to our work. Brief discussions of how they well they seemed to work are also included.

5.1 Metric I: Lines of Code

5.1.1 Description

Using Lines of Code (LOC) as a metric has a long tradition. Several papers and books describe this. Fenton [20] gives a good and thorough explanation of how LOC has been used and can be used.

Several variations of this metric exist, and it is important to be precise about the definition of a Line of Code. Some people measure every line, while others don't include blank lines or comments. Others only count lines containing an operator. We have chosen to count every line except blank lines and comments. Different programmers have different programming styles. Some like their code to be compact, others use a lot of whitespace. Some comment a lot, others more sparsely. Eliminating blank lines and comments makes the result indifferent to this. To further ensure that different programming styles don't influence the result we have used the Eclipse editor's built-in source formatter to format the code.

Although LOC is a widely known and used metric, there are many who believe that it is a poor indicator of anything except size. We have chosen to use LOC as one of our metrics anyway. This way, we can see if LOC can be useful to describe the differences between AOP and OOP, a novel area for this metric. LOC is such an easily assessable metric that it would be nice to be able to use it.

Instead of merely looking at the total number of LOC, we have chosen to measure how the growth in LOC evolves as we add our new features. As authentication is a one-time process, it doesn't have such a growth. But when it comes to authorization, we can distinguish between the code that is necessary to make authorization work at all and the code that is necessary each place we want to add authorization in the code. Measuring it this way can give us a formula that describes the evolution of LOC: $LOC_{total} = x + n * y$. This makes it possible not only to see what required the least amount of LOC for the example I have used, but probably also for other projects.

In the evaluation of the results, a small number for x and y is the best. In a tradeoff between x and y , the most important factor is y , that describes how many LOC you must add for each authorization-point. Especially for large projects, this is the number that will influence the total the most.

5.1.2 Results

In the OOP version, the authentication is 23 LOC, while the authorization is between 12 and 15 LOC, depending on where in the code we add the authorization checks. The mean LOC is 14. This gives us the formula $LOC_{total} = 23 + n * 14$.

In the AOP version, the authentication and authorization code is 77 LOC before we add any authorization checkpoints at all. Each checkpoint takes either 1 or 6 LOC to accomplish. The checks to limit the GUI, which were more difficult and complex, take 6 LOC, the others only 1 LOC. The mean is 3 LOC. This gives us the formula $LOC_{total} = 77 + n * 3$.

5.1.3 Discussion

We see that there is a great difference in the results between the AOP and the OOP version. Although the AOP version has a greater initial LOC value, the cost of increasing the amount of authorization checkpoints is minimal compared to the OOP version. The AOP version is therefore favorable, in accordance with what we wrote in the description of this metric.

We believe this is an indication that authorization should be treated as a crosscutting concern, and thus is suited to be put in an aspect. Not only is it possible to decrease the LOC you must write for each authorization checkpoint, but the lines you need to write are pretty much the same each time. This observation is not directly given by the LOC numbers, but the reason why we need so few LOC to add a new authorization checkpoint

using AOP is just this. Because the code is so similar it is easy to refactor it out into an advice.

It is possible that the LOC values for the AOP version could have been additionally decreased. We have made an abstract aspect that has two concrete subspects, one for the client code and one for the server code. This is in conformity with good programming practices, but increases the LOC measures. The fact that we used the authorization checks not only to limit what code the user can run, but also to limit what GUI elements the user can see, presented us with some problems. We couldn't reuse the same advice for these two situations, because they differ in the way unsuccessful authorizations should be treated. We had to make two very similar advice to cover these two situations. It is possible that we could have refactored some of the shared code into a common method, thus decreasing the LOC value.

Another important aspect of the LOC metric, that we would like to mention but have not put a great deal of emphasis on, is *where* in the code the new lines are entered. In the AOP version, not a single line of code in the original program has been altered, and not a single line of code has been added to any of the original objects. In the OOP version, several classes has been modified. The fact that it is possible to add JAAS to the Pet Store application without altering the original sourcecode is an addition suggestion that authentication and authorization indeed represents orthogonal functionality, and thus is suited to be implemented by AOP.

5.2 Metric II: Code complexity

5.2.1 Description

It is a common belief that there is a link between the structure of an application's code, and the quality of the product. Many researchers have tried to measure the structural properties of software. The intention has been to establish a link between these measurements and properties like how easy it is to test and maintain a product, failure rate and other external attributes.

As we wrote in the previous chapter, the Common Criteria is also concerned about code complexity:

Design complexity minimisation contributes to the assurance that the code is understood -- the less complex the code in the TSF, the greater the likelihood that the design of the TSF is comprehensible. Design complexity

minimisation is a key characteristic of a reference validation mechanism.
[28]

CC present several requirements about how developers should minimize complexity:

Developer action elements

- ADV_INT.3.1D The developer shall design and structure the TSF in a modular fashion that avoids unnecessary interactions between the modules of the design.
- ADV_INT.3.2D The developer shall provide an architectural description.
- ADV_INT.3.3D The developer shall design and structure the TSF in a layered fashion that minimises mutual interactions between the layers of the design.
- ADV_INT.3.4D The developer shall design and structure the TSF in such a way that minimises the complexity of the entire TSF.
- ADV_INT.3.5D The developer shall design and structure the portions of the TSF that enforce any access control and/or information flow control policies such that they are simple enough to be analysed.
- ADV_INT.3.6D The developer shall ensure that functions whose objectives are not relevant for the TSF are excluded from the TSF modules.

There are several different approaches to measuring code complexity. The one we have chosen to use is McCabe's cyclomatic complexity measure. McCabe's work was originally done in 1976, but has withstood the test of time. His work has been incorporated into several open source measuring tools, such as CheckStyle (<http://checkstyle.sourceforge.net>) and Metrics (<http://metrics.sourceforge.net>).

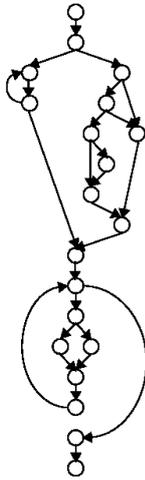
Although Fenton in his discussion is skeptical to the use of the cyclomatic number as a general complexity measure, he does however believe it to be a useful indicator of how difficult an application will be to test and maintain. And this is, after all, what we are looking for.

A number of studies have confirmed the usefulness of the cyclomatic number. Grady reported from a study involving 850000 lines of code that there was a correlation between a module's cyclomatic number and the number of updates required to the module. Several other studies are presented in [25].

Cyclomatic complexity is based on the structure of an application's control flow graphs. A control flow graph describes the logic structure of

software modules. Each possible execution path of a software module has a corresponding path from the entry node to the exit node of the module's control flow graph.

An example of Control Flow Graph is show in below:



Cyclomatic complexity is defined to be $e - n + 2$, where e and n are the number of edges and nodes in the control flow graph. Cyclomatic complexity is also known as $v(G)$, where v refers to the cyclomatic number in graph theory and G indicates that the complexity is a function of the graph. $v(G) = e - n + 2$.

There are simpler ways to compute the cyclomatic complexity than counting all edges and nodes. There are several automatic tools that can do the job for you. There is also another way if you measure by hand, you can count the number of decision predicates. If all decision predicates are binary and there are p decision predicates, then $v(G) = p + 1$. A binary decision predicate appears on the control flow graph as a node with exactly two edges flowing out of it. The metric application "Metrics" count the following statements and constructs as binary predicates in Java: `if`, `for`, `while`, `do`, `case`, `catch`, the ternary operator `?:`, and the conditional operators in expressions; `&&` and `||`.

We count the same way as "Metrics", with one exception. The reason the `&&`-operator is counted, is that in Java (as in C++ and other programming languages), the second part of an expression combined with `&&` will be skipped if the first part is evaluated to be false. This is called short circuiting, and can lead to two different paths of execution.

This is fine in regular object-oriented Java programming, but not when it comes to aspect-oriented programming and pointcuts. In a pointcut, the expression is not a part of a program flow, it is used to describe and capture a program flow. The code isn't executed in the regular sense. This means that an expression can evaluate to both true and false, depending on the joinpoint you look at.

Take the following code as an example:

```
public pointcut doPostOperation(ServletRequest req )
    : call( BufferedReader ServletRequest.getReader() )
    && within( ApplRequestProcessor )
    && target( req );
```

This pointcut consists of three parts. The first part captures every call to `ServletRequest.getReader()`. There are only a few joinpoints in the code this matches, but there are thousands of joinpoints it doesn't match. The second part limits the possible joinpoints to those inside the class `ApplRequestProcessor`. This reduces the impact of the pointcut, and helps clarify which joinpoints are affected. This clearly doesn't add complexity, if anything it reduces it. The third part of the expression isn't evaluated at all; it is included only to capture the `ServletRequest`-object in the code, so that it can be used in the advice. The conclusion of this little discussion is that the `&&`-operator in a pointcut doesn't short-circuit like the regular `&&` in Java, and doesn't add to the cyclomatic complexity.

According to the documentation following the software metrics analyzing tool "CheckStyle", the cyclomatic complexity of a module can be evaluated according to the following limits:

1-4 is good, 5-7 is considered ok, 8-10 means you should consider refactoring your code, and >10 means you definitely should refactor your code.

The limit of 10 is used by the "Metrics" tool as well.

According to [26], cyclomatic complexity should not be aggregated for an entire class. Because of the fact that every method has a minimum cyclomatic complexity of one, a class with many methods would automatically have a high score, although it wouldn't necessarily be very complex. An example of this is a typical domain object that adheres to the rules of JavaBeans. An object like this would have several getter- and setter-methods that would contribute greatly to the total cyclomatic complexity. The correct way to evaluate cyclomatic complexity is on a per-method basis.

5.2.2 Results

The following table shows the results of measurements:

Class and method affected	AOP	OOP	Original
ApplRequestProcessor			
processRequest	0	+1	8
getChartInfo	0	0	6
updateOrders	0	0	5
getOrders	0	0	4
HttpPetStoreProxy			
doHttpPost	0	0	8
PetStoreAdminClient			
createUI	0	+2	1
createMenuBar	0	+1	1
createToolBar	0	+1	1
AboutAction.actionPerformed	0	+1	1
SubjectHolder			
getAuthenticatedSubject		3	
NullCallbackHandler.handle		1	
AbstractPetStoreJAASAspect			
before : authOperations	3		
around : authorizationOperations	2		
before : authorizationOperations	1		
Authenticate	1		
NullCallbackHandler.handle	1		
PetStoreJAASServerAspect			
around : doPostOperation	2		
getPermission	1		
pointcut authorizationOperations	3		
pointcut doPostOperation	1		
PetStoreJAASClientAspect			
getPermission	1		
around : restrictingMenu restrictingToolBar *	2		
before : restricting *	1		
around: httpPostOperation	2		
pointcut authOperations	1		
pointcut authorizationOperations	1		
pointcut httpPostOperation	1		
pointcut restrictingToolBar	1		
pointcut restrictingMenu	1		

* Not including the || in the anonymous pointcut

Total increase in cyclomatic complexity for OOP: 10
Total increase for AOP, not including pointcuts: 17
Total increase for AOP, including pointcuts: 17 + 9

5.2.3 Discussion

The results we get from this metric need some interpretation. We see that the total increase in cyclomatic complexity is higher when we use AOP. But on the other hand no method in the AOP version has a higher cyclomatic complexity than 3, which is a very good number. In the OOP version, where the authorization code is placed inside the core modules, it heightens the cyclomatic complexity of these modules. This can help bring an already high number even higher, as the cyclomatic complexity is increased by one or two for each authorization point, depending on the operation.

The increased total complexity in the AOP version can be accounted for by the large increase in number of methods. While the OOP version only has two new methods (both in the SubjectHolder class), the AOP version has eleven (not including the pointcuts). As the minimum cyclomatic complexity for a method is 1, this large number of methods has a clear impact on the result. As we write in the description of this metric, cyclomatic complexity should not be aggregated for an entire class, but should be evaluated only on a method-basis.

As a conclusion, we clearly favor the AOP version. It doesn't affect the core methods in a negative way, and every new method is kept at a low and manageable level when it comes to cyclomatic complexity. Cyclomatic complexity is not meant to be calculated at a package level or even a class level, but at a method level. And the overall cyclomatic complexity of the methods involved is lower in the AOP version.

5.3 Metric III: Chidamber and Kemerer Metrics

5.3.1 Description

Chidamber and Kemerer's paper [4] about software metrics is an important paper, and the metrics they proposed has been widely used, and still are. Chidamber and Kemerer base their work on a set of principles proposed by Bunge and later applied to object-oriented systems by Yand and Weber. "The world is viewed as being composed of substantial individuals that possess a finite set of properties. Collectively, a substantial individual

and its properties constitute an object. A class is a set of objects that have common properties, and a method is an operation on an object that is defined as part of the declaration of the class.” [20, p. 318]. Chidamber and Kemerer use these notions to define a number of metrics that are claimed to relate to some of these attributes:

- Weighted Methods Per Class (WMC)
- Depth of Inheritance Tree (DIT)
- Number of Children (NOC)
- Coupling between objects (CBO)
- Response For a Class (RFC)
- Lack of Cohesion in Methods (LCOM)

Not all the metrics are useful for us in our scenario. We do not create a lot of new classes in neither the AOP version nor the OOP version, because we do not build a system from the ground up. Especially in the OOP version, we mostly add code to existing methods. This means that some of the C&K metrics, like Depth of Inheritance Tree and Number of Children doesn’t change. We have also excluded the Lack of Cohesion in Methods metric, as it didn’t give us much result. There are too few variables used in our code.

The three remaining metrics are useful to us, and we present them below. Tsang et al. [19] have used these metrics to evaluate a program implemented in both OOP and AOP before us, and we discuss their adaptation of the metrics along with the introduction to the metrics.

We have not found any literature saying how large these numbers should be or how large they shouldn’t be. This will make it more difficult for us to declare a winner, and we will merely discuss and try to interpret the results.

5.3.1.1 Weighted Methods per Class (WMC)

Weighted Methods per Class is a measure of the number of methods implemented within a class.

Tsang adapt WMC for use with AOP by counting aspects as classes and advice as methods. We agree with this adaptation, and have chosen to do the same. However, we have chosen to use McCabe’s cyclomatic complexity to weight each method. This is a quite common approach, used by both the source analyzing tool “Metric” and by other researchers. [20 p. 319] Tsang does not do this, they don’t seem to use any sort of weight-system. In our opinion, this deteriorates their results.

5.3.1.2 Coupling Between Objects (CBO)

Coupling Between Objects is a count of the number of other classes from which elements are used, i.e calls or attribute access between classes.

The adaptation Tsang et al. made to this metric to make it suitable for AOP is in our view far from perfect. They considered aspects coupled to classes only if the aspects explicitly name the classes. The joinpoint call(* *(..)) is therefore not coupled to any other class, although it will match every other class. The joinpoint call(org.hig.Test.doTest(..)) is coupled to the Test class in the org.hig package. In practice, this might be an approach that works fairly well. But it has some obvious flaws that we would like to discuss here.

Firstly, in AOP one must distinguish between the caller and the callee. The caller is the object making the call, the callee is the object being called upon. The call-joinpoint is on the caller side, not the callee side. If you want to weave code on the callee side, you should use the execute-joinpoint. In most cases, the effect will be the same, but not always. This means that the coupling should not necessarily be to the class mentioned in the pointcut, because that will probably be the callee in both cases. In the call(org.hig.Test.doTest(..)) joinpoint the advice is not woven into the Test-class, but into whatever class it is that calls the Test-class.

Secondly, in some cases coupling may seem to increase if we use Tsang's definition, although it actually decreases. This can be shown in the following two pointcuts:

```
public pointcut restrictingToolBar_version1()
    : call(* java.awt.Container.add(java.awt.Component+));

public pointcut restrictingToolBar_version2()
    : call(* java.awt.Container.add(java.awt.Component+));
    && withincode(* PetStoreAdminClient.createToolBar(..));
```

These two pointcuts are the same, except that the last one has added an extra line. This line would, if we follow Tsang's rules, add a coupling to a new class from the aspect. But the point behind using the withincode construct is to limit the classes affected by the pointcut. The first call-joinpoint can match code in several classes. Because we only wanted to weave our advice into the a couple of specific places in the PetStoreAdminClient class, we added the second line. This helps us limit the number of joinpoints the pointcut matches, and should therefore not

increase the coupling. Our opinion is that this rule is in violation of the measurement theory as described by Fenton [20].

Coupling is clearly a difficult metric to measure. In OOP, coupling is well defined and easy to measure. But AOP is by its nature a more fluid and less stringent way to program. Tsang's solution to how we should define coupling is probably based on the fact that it is easy to use: If a classname is included, we'll count it. This can be done manually, by merely looking at the code. How easy a metric is to assess is an important factor, so we are not trying to ridicule this in any way. We believe however that their definition is not precise enough, and should be replaced. Our opinion is that the CBO metric should reflect where the aspect is being weaved into the core code. The distribution of the AspectJ compiler also includes a tool for displaying the effect a pointcut has on the core code. This tool, or a similar one, can give us the answer to how many classes are affected by a given pointcut.

But this also has its limitations. It can make it impossible to measure coupling by looking at the aspect alone, we must always examine the entire system together. If we introduce new core classes into a system, the number of couplings may increase, and the same aspect can have a different number of couplings in two systems. In our own code however, this has not been a problem. Our pointcuts are defined in such a way that it is easy to see which classes they affect.

The approach we have chosen in regard to the first problem we discussed about Tsang's definition, is to treat call and execute joinpoints differently. In an execute joinpoint, code is woven into the actual method that is executed. This means that no other methods are involved, and the coupling between the aspect and the core code should only be to the class that contains the executed method. In a call joinpoint, code is woven into the method that calls the method mentioned in the joinpoint. The code is inserted on the caller side. This creates a coupling from the aspect to both the class being called, and the caller class. Our reasoning for this is that it is the caller class where the weaving takes place, but it is the call to class mentioned in the joinpoint that decides where in the caller class the code should be woven.

This has been a long discussion. It is important to decide how to adapt the CBO metric to AOP. However, we want to point out that pointcuts are not the only source of coupling. Every use of another class from within an object creates a coupling. To count these we use the import-statements as a starting point, and also check to see if classes from the same package are used as well.

5.3.1.3 Response For a Class (RFC)

Response For a Class is the number of methods that can potentially be executed in response to a message received by an object of a class. This includes every method in the object itself, as well as every method in other classes directly called by a method in the object. (Only the first level is counted, we don't traverse further down.)

Again, we choose to count as Tsang does. They count invocations from core code to aspects when calculating RFC. We know when advice in the aspects are called through the pointcut definitions. We see that similarly to our needs when calculating CBO, we need to know exactly which classes are affected by an aspect. This is fortunately no problem in our code, but generally speaking, a tool like the aspect browser that comes with AspectJ might come in handy.

5.3.2 Results

5.3.2.1 Weighted Methods per Class

The numbers shown are the increase (λ) for each class, not the total number.

	OOP	AOP
ApplRequestProcessor	1	0
HttpPostPetStoreProxy	0	0
PetStoreAdminClient	5	0
SubjectHolder	4	NIL
AbstractPetStoreJAASAspect	NIL	8
PetStoreJAASServerAspect	NIL	3
PetStoreJAASClientAspect		6

5.3.2.2 Coupling Between Objects

Here the numbers are the total number for each class.

	Original	OOP	AOP
ApplRequestProcessor	27	33	27
HttpPostPetStoreProxy	22	24	22
PetStoreAdminClient	24	31	24
SubjectHolder	NIL	6	NIL
AbstractPetStoreJAASAspect	NIL	NIL	10
PetStoreJAASServerAspect	NIL	NIL	13
PetStoreJAASClientAspect	NIL	NIL	19

5.3.2.3 Response For a Class

Here the numbers presented are again only the increase (λ), not the total number.

	OOP	AOP
ApplRequestProcessor	10	4
HttpPostPetStoreProxy	4	1
PetStoreAdminClient	9	4
SubjectHolder	7	NIL
AbstractPetStoreJAASAspect	NIL	15
PetStoreJAASServerAspect	NIL	11
PetStoreJAASClientAspect	NIL	16

5.3.3 Discussion

We see similar results here as we did when we calculated McCabe's cyclomatic complexity. The core classes have lower numbers in the AOP version than in the OOP version, but the aspects in the AOP version have so high numbers that it is difficult to select a winner. It would probably have been better if the application we worked with was bigger, and we had to implement more code to add JAAS everywhere we wanted. The results we get with this metric are not very decisive. However, we tend to like the more balanced results we get from the AOP version. Although the total number (if all results are added) is higher, we have no extremely huge classes. Responsibility, methods and coupling is more evenly distributed among the classes in the code.

6. CONCLUSIONS

We see through our work that AOP shows merit when it comes to implementing authentication and authorization. Both the LOC metric and the complexity metric suggest that the AOP version has an advantage, especially for large applications. The last metric is more open for interpretation, but the combined result gives a clear indication that the AOP version is the best alternative. Its scores seem more balanced, and it doesn't affect the core code in a negative way but distributes the load to several classes/aspects. If we look closer at how the metrics are calculated, we also believe that if the application we were adding JAAS to had been greater, AOP's advantage would probably be easier to see. AOP also permits us to organize the Trusted Computing Base in fewer files with clearer

boundaries, making it easier to get an overview of the code involved and easier to perform a code review.

The fact that the core code is virtually unaffected in the AOP version is shown in all three metrics, and this is perhaps the best suggestion that authentication and authorization in the form of JAAS is well suited for aspect-oriented programming.

We have also seen that AOP, especially dynamic AOP, requires you to pay good attention to the code you program, deploy and run. AOP makes it possible for an intruder to sneak malicious code into your application, and you need good routines to prevent this from happening.

7. FUTURE WORK

Our results suggests that AOP is indeed a viable and good choice if you want to implement JAAS in you application. But because our codebase was not all that big, further studies should be performed. Although it will be both time-consuming and require quite a lot of resources, a large scale case study or experiment would be very useful. This can show if we are right in our predictions that AOP's advantage over OOP will increase as the codebase increases.

What we would suggest is to find a college or university that teaches AOP to its students. By using students that are familiar with AOP, a large scale experiment could be performed. A testing framework should be established, possibly using a tool like JUnit (<http://www.junit.org>). By giving the students a predefined application they should add JAAS to, and by making a set of unit tests in JUnit that can confirm when the necessary code has been added, it should be possible to perform such an experiment quite easily. Tools that extracted the necessary metrics from the students' code should also be made, as it would be a very tedious process to do manually. The results can be correlated with the students' grades and the time invested in the task to see how this factors influence the results.

We would also like to see better support for metrics covering AOP in the various open source tools out there. Both "Metrics", "CheckStyle" and "JDepend" are great tools, but they only support regular OOP. We had to do most of our measurement by hand, and this was a tedious process. A key to success for everything new when it comes to programming is tool support. AspectJ already comes with some great tools and is supported in several of the major IDEs, but support for AOP is lacking in most other tools.

8. REFERENCES

- [1] M. J. Yuan and N. Richards. Lightweight Aspect-Oriented Programming: Putting the Interceptor pattern to work. *Dr. Dobbs's Journal*, no. 351, August 2003.
- [2] N. Lesiecki. Improve modularity with aspect-oriented programming. 1 January 2002. [URL: <http://www-106.ibm.com/developerworks/java/library/j-aspectj/>]
- [3] R. Laddad. I want my AOP!, Part 1. [URL: <http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html>]
- [4] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, vol. 20, no. 6, 1994
- [5] B. Vanhaute and B. De Win. AOP, Security and Genericity. 12. July 2001. [URL: <http://www.cs.kuleuven.ac.be/cwis/research/distrinet/public/showperson.php?ID=2>]
- [6] J. Viega and J. Voas. Can Aspect-Oriented Programming Lead to More Reliable Software? *IEEE Software*. November/December 2000.
- [7] J. Viega, J.T. Bloch and P. Chandra. Applying Aspect-Oriented Programming to Security. *Cutter IT Journal*, vol. 14, no. 2, February 2001.
- [8] R. Laddad. *AspectJ in Action*. Manning Publications Co, 2003.
- [9] B. De Win, B. Vanhaute and B. De Decker. How aspect-oriented programming can help to build secure software. *Informatica –Ljubljana–*, vol. 26, no. 2. 2002.
- [10] M. Mezini, K. Ostermann and R. Pichler. Component Models and Aspect-Oriented Programming. 2001. [URL: <http://www.st.informatik.tu-darmstadt.de:8080/lehre/ws01/sctoo/materials/aj-aop.pdf>] (Accessed 15. December 2003.)
- [11] D. Palmer. Dynamic Aspect-Oriented Programming in an Untrusted Environment. Workshop on Foundations of Middleware Technologies, part of the International Symposium on Distributed Objects and Applications, 2002.
- [12] A. Popovici, G. Alonso and T. Gross. Just-In-Time Aspects: Efficient Dynamic Weaving for Java. Proceedings of the 1st international conference on Aspect-oriented software development, AOSD 2003, Boston, USA, 2003.
- [13] G.C. Murphy, R.J. Walker, E.L.A. Baniassad, M.P. Robillard, A. Lai and M.A. Kersten. Does Aspect-Oriented Programming Work? *Communications of The ACM*, vol. 44, no. 10, October 2001.
- [14] B. De Win, F. Piessens, W. Joosen and Tine Verhanneman. On the importance of the separation-of-concerns principle in secure software engineering. Workshop on the Application of Engineering Principles to System Security Design, Boston, Usa, November 6-8 2002.

- [15] J. Viega and G. McGraw. Building Secure Software. Addison-Wesley Professional Computing Series, 2001.
- [16] B. Kitchenham, L. Pickard and S.L. Pfleeger. Case studies for method and tool evaluation. Software, IEEE, vol. 12, no. 4, July 1995, p. 52-62. [URL: <http://www.idi.ntnu.no/emner/mnfit365/artikler/s4052.pdf>]
- [17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier and J. Irwin. Aspect-Oriented Programming. Proceedings of the European Conference on Object-Oriented Programming, Finland, June 1997.
- [18] A. A. Zakaria and Dr. H. Hosny. Metrics for Aspect-Oriented Software Design. Proceedings of the 1st international conference on Aspect-oriented software development, AOSD 2003, Boston, USA, 2003.
- [19] S. L. Tsang, S. Clarke, E. Baniassad. Object Metrics for Aspect Systems: Limiting Empirical Inference Based on Modularity. Submitted to ECOOP 2004, Oslo, Norway, 14-18 June 2004.
- [20] N.E. Fenton and S.L. Pfleeger. Software Metrics. 2nd Revision edition. Brooks Cole, 1998
- [21] I. S. Welch and R. J. Stroud. Re-engineering Security as a Crosscutting Concern. The Computer Journal, vol. 46, no. 5, 2003
- [22] J. Zhao. Towards A Metrics Suite for Aspect-Oriented Software. Technical-Report SE-2002-136-25, Information Processing Society of Japan (IPSJ), 2002.
- [23] B. Dufour, C. Goard, L. Hendren, C. Verbrugge, O. de Moor and G. Sittampalam. Measuring the Dynamic Behaviour of AspectJ Programs. Accepted for OOPSLA '04 - revisions pending. OOPSLA 2004, Vancouver, British Columbia, October 24-28, 2004.
- [24] T. McCabe. A Complexity Measure. IEEE Transactions on Software Engineering, December 1976.
- [25] A. H. Watson and T. J. McCabe. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. NIST Special Publication 500-235. September 1996.
- [26] Andrew Glover. Code Improvement Through Cyclomatic Complexity [URL: <http://www.onjava.com/pub/a/onjava/2004/06/16/ccunittest.html?page=1>]
- [27] The Common Criteria. [URL: <http://www.commoncriteriaportal.org>]
- [28] Common Criteria for Information Technology Security Evaluation. Part 3: Security Assurance Requirements. Ver. 2.2, rev. 256. January 2004. [URL: <http://www.commoncriteriaportal.org/public/files/ccpart3v2.2.pdf>]