A Process to Combine AOM and AOP: A Proposal Based on a Case Study

Jingyue Li
Norwegian University of Science and Technology
Sem Sælands vei 7-9
NO-7034 Trondheim, Norway
+47, 73598716
Jingyue@idi.ntnu.no

Siv Hilde Houmb
Norwegian University of Science and Technology
Sem Sælands vei 7-9
NO-7034 Trondheim, Norway
+47, 91307714
sivhoumb@idi.ntnu.no

Axel Anders Kvale
Norwegian University of Science and Technology
Sem Sælands vei 7-9
NO-7034 Trondheim, Norway
+47, 93034377
axelkv@stud.ntnu.no

ABSTRACT
Traditional object-oriented programming (OOP) paradigm focused on structuring systems into distinguished objects that work together to realize a system. However, when dealing with non-functional or quality requirements, such as security and fault tolerance, these are not easily structured into separate objects, but do rather crosscut a set of objects. Aspect-oriented programming (AOP) separate crosscutting concerns into single units called aspects. Aspect-oriented modelling (AOM) techniques allow system developers to design and verify an aspect-oriented system on the modelling level. During a case study of re-engineering an object-oriented system using aspect-oriented programming, we learned that well-designed aspect-oriented modelling (AOM) is essential to the success of aspect-oriented system. We also learned that current aspect-oriented programming tools (AOP) pose limitations on the design of an ideal aspect-oriented model.

Based on lessons learned from this study we propose a process that handles aspects at two levels, both at the modelling level (AOM) and the programming language level (AOP). At the AOM level, aspects are identified and weaved together by AOM weaving to verify and do trade-off between different mechanisms. However, models are not woven together for the purpose of code generation based on a combined model. The actual weaving is done by the AOP compiler.

Keywords
Aspect-Oriented Development (AOD), Aspect-oriented Modeling (AOM), Aspect-Oriented Programming (AOP), Model-Driven Development (MDD)

1. INTRODUCTION
Aspect-oriented programming (AOP) is claimed to be able to increase the maintainability of systems compare to Object-oriented programming (OOP) [5, 16]. In COTS component-based development, the invocation of COTS component functionalities or methods are scattered in the system. If crosscutting concerns in glue-code can be separated into aspects, it will be easy to understand and change the system. To empirically investigate how to build an easy-to-change COTS component-based system using AOP we performed a case study where we compared the easy of software evolution in an object-oriented aspect-oriented version of the same system. Results from this study indicate that benefits of AOP cannot be acquired without a good aspect-oriented design. Furthermore, limitations of current AOP tools pose some difficulties related to implementing a good design. Based on lessons learned from this study we propose an approach that combines the AOM and AOP. AOM is used in the requirement and design phase to ensure a good aspect-oriented design. Conflict testing and trade-off analysis between aspects are performed on the modelling level, which includes a primary model and several aspect models. These models are implemented and weaved together using AOP and AOP weaver.

The reminder of this paper is organized as following. Section 2 gives a short introduction to AOP and AOM. Section 3 describes the case study and the lessons learned. Section 4 presents the combined AOM and AOP process, while Section 5 gives an example. Conclusions and future work are presented in Section 6.

2. BACKGROUND
Aspect-oriented development (AOD) emphasizes the separation of concerns and is designed to handle complex structures. Both AOP and AOM are part of the AOD paradigm.

2.1 Aspect-Oriented Programming (AOP)
Aspect-oriented programming (AOP) is a new technology for separation of crosscutting concerns into single units called aspects. An aspect is a modular unit of crosscutting implementation. It encapsulates behaviours that affect multiple classes into reusable modules. Aspectual requirements are concerns that introduce crosscutting in the implementation. Typical aspects are synchronization, error handling or logging. With AOP, each aspect can be expressed in a separate and natural form, and can then be automatically combined together into a final executable form by an aspect weaver. As a result, a single aspect can contribute to the implementation of a number of procedures, modules or objects, increasing reusability of the codes. The differences between AOP and traditional programming are shown in Figure 1. Compared to traditional approaches AOP allows separation of crosscutting concerns at source code level. The aspect code and other part of the program can be woven together by an aspect weaver before the program is compiled into an executable program.

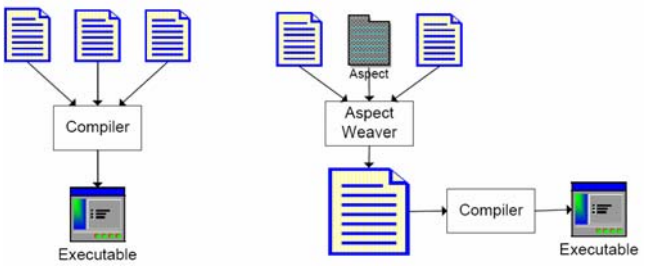

Figure 1. Comparing to the traditional approach.

An AOP language has three critical elements for separating crosscutting concerns: a join point model, a means of identifying join points, and a means of affecting implementation at join points [5].

2.2 Aspect-Oriented Modeling (AOM)

Aspect-oriented modelling (AOM) techniques allow system developers to address crosscutting and quality objectives such as security separately from core functional requirements during system design [8]. An aspect is a pattern of structure and behaviour such that it is a crosscutting realization of common structural and behaviour characteristics [17]. An aspect model consists of a UML class diagram template (or other structural diagrams of UML) and one or more interaction diagrams templates. The structural diagram templates generate structural diagrams that are used to describe the structure of the system. Interaction diagram templates are used to generate interaction diagrams that describe how elements in the distributed structures interact to realize the desired behaviour.

An aspect-oriented design model consists of a set of aspects and primary models. An aspect model describes how a single objective is addressed in the design, while the primary model addresses the core functionality of the system as given by the functional requirements.

In AOM, one makes use of composing rules for weaving aspect models with the primary model. These rules are stored separately from the aspect and the primary model, which makes both the aspect models and the rules reusable. The aspects and the primary model are composed before implementation or code generation. Composition is most often done manually, but there exists tools that automate part of the composition. Figure 2 gives an overview

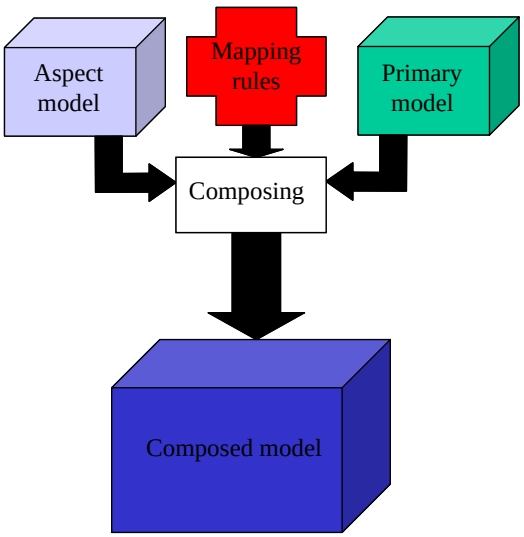

Figure 2. A general overview of the AOM approach

of AOM.

In [14] Rashid, Moreira, and Arauho, looks into aspects in the requirement capturing phase and target multidimensional separation beginning early in the software cycle as part of the early aspects initiative. Their work supports modularization of crosscutting properties at the requirements level and the main aim is to support early trade-offs. Other related AOM approaches are [1, 3, 4, 6, 7, 8, 15], where [6, 7, 8] targets security issues in particular.

3. CASE STUDY

After an result of an empirical study on COTS (Commercial-Off-the-shelf) component based development in Norwegian IT industries we discovered a three steps COTS component selection process; (1) selecting handful COTS components by Internet searching; (2) selecting 2 to 3 possible candidates based on some key issues; (3) integrating these possible candidates into intended environment and test them in order to select one [12]. To integrate COTS components into possible future system instead of testing them individually, a lot of glue-code must be produced. A challenge, however, is to be able to reuse glue-code in such a way that no much extra effort need to be spent on changing testing environment from one COTS component to another.

AOP is claimed to be enable separation of concerns and increase the maintainability of the system comparing to object-oriented programming. The initial motivation of this study is to investigate whether aspect-oriented programming can help to increase the glue-code reusability in a COTS component-based system.

3.1 Case study design

The case study includes three steps:

Step 1: We re-engineered an object-oriented application using AOP. Glue-codes relevant to some components were implemented using AspectJ (version 1.1 [19]).

Step 2: We used other COTS components to replace some crosscuting components in both the object-oriented and aspect-oriented version of the system.

Step 3: The number of Line-of-code and classes that needed to be changed in the object-oriented and aspect-oriented version during component replacement were measured and compared.

The system chosen for the study was an open source Java Email Server (i.e., JES server [18]). It is a stand-alone java application that was built by using Object-oriented based development. Although the application is an open source system, we treated all components as COTS components (i.e. no source code was modified) in this study.

3.2 Lessons learned

During the re-engineering of the system we discovered several challenges and difficulties related to AOP. These issues need to be solved to enhance the assumed benefits of AOP (i.e. better maintainability comparing to object-oriented programming).

The key lessons learned from the case study were:

- **A good aspect-oriented design is essential to achieve the benefits of aspect-oriented programming**

- **Limitations of aspect-oriented programming tools should be taken into consideration in aspect-oriented design.**

3.2.1 A good aspect-oriented design is essential to achieve the benefits of Aspect-oriented programming.

In the process of re-engineering the JES server we used source code reading to identify the proper aspect. A typical ideal aspect example is logging. Logging objects are inserted into every class that makes us of this feature. If we implement the logging functionality using aspect, the aspect can easily trap every join point and log all the run-time information available. This makes it is easy to implement logging into several classes with only a minimum of code lines and without changing any of the original classes. However, there are a few practical concerns that arises. When logging is inserted into each class in the traditionally OOP-way, each line of logging is usually very accurate. The developer can choose to write whatever information to the log to get a reasonable clue of what the system did (or failed to do). With the general logging using AOP this cannot be easily accomplished. The logging will be limited to the information provided by the joinpoints (name of the function, name of the enclosing function, arguments, class name etc.).

To implement accurate logging in AOP we had to define each pointcut and treat these joinpoints individually. The implementation is shown in the following code:

```
//defining joinpoint #1

private pointcut PC1(LogInterface li, int x, int y) :

        this(li) && args(x,y) && execution(public void Function1(int x, int y));

//logging in joinpoint #1

after(LogInterface li, int x, int y) returning: PC1(li, x, y){

        li.log.info("Changed X and Y to (" + x + "," + y + ")");

}

//defining joinpoint #2

private pointcut PC2(LogInterface li, String s) :

        this(li) && args(s) && execution(public void Function1(String s));

//logging in joinpoint #1

after(LogInterface li, String s) returning: PC1(li, s){

        li.log.info("Changed name to " + s);

}
```

The result is that we have to deal with several and sometimes quite complex joinpoints to retrieve the required information and write more lines of code to ensure that the aspect-oriented version has the same functionalities as the object-oriented version. Furthermore, we need to change more lines of code in the AOP version when we used another logging COTS component to change the current logging component in JES (see step2 in case study design in chapter 3.2). This means that we loose the benefit (and strengths) of using AOP.

Although logging is regarded as an ideal aspect in some other systems [2] we experience otherwise in this case study. The basic reason is that the system was designed based on OOP thinking with no proper aspect-oriented design from beginning. Therefore, it is hard to re-engineer the system into an ideal aspect-oriented system using an object-oriented design.

3.2.2 Limitations of aspect-oriented programming tools should be taken serious consideration in aspect-oriented design

When we implemented the aspect-oriented system using AspectJ 1.1 some unexpected limitations of AspectJ 1.1 made it difficult to implement part of the design.

Static problem

Intertyping is a functionality of AspectJ 1.1 that enables you tom insert a reference to an object or variables into a class from an aspect like following.

```
//Intertyping a log object into the class User

private Log com.ericdaugherty.mail.server.info.User.log;
```

When interpying references into several classes, an elegant design is to create an interface and let all classes implement this interface. By utilizing an interface when intertyping references, we can get a common handle to the classes that we can use when accessing the log-object.

When we implemented the system re-engineering using this design we found that the current version of AspectJ (version 1.1) does not support intertype declarations of static members to multiple classes. The static log must be intertyped into each class using a static log as follows:

```
//Intertyping a static log into the class Message

private static Log com.ericdaugherty.mail.server.info.Message.log
=LogFactory.getLog(com.ericdaugherty.mail.server.info.Message
.class);
```

When intertyping directly into a class instead of using an interface the benefit of getting a common handle to the classes disappears.

Although some previous study mentioned the static limitations of AspectJ (version 1.1), we did not take this into serious consideration since we made the design before looking into current available tools.

Accessing variable inside a block statement

A pointcut can create a reference to all variables used in a pointcut. Possible variables are:

- The object making the call (this)
- The object receiving the call (target)
- Variables passed as parameters to the method
- The returning value of the method

If other variables is needed several pointcuts is necessary in order to get references to these variables.

```
public void DoSomething(String s){

        //do something

        EmailAddress address = new EmailAddress(s);
```

```
        User user = new User(address); //This is the joinpoint we want to trap
        //do something more
}
```

If we want to access the input strings when the user is created in the above code we need to combine several pointcuts.

```
//Pointcut picking out the extra variable String s
private pointcut DoSomething(String s) :
execution(void DoSomething(String)) && args(s);
//Pointcut picking out the joinpoint and the variables user and address
private pointcut NewUser(User user, EmailAddress address) :
target(user) && call(User.new(EmailAddress)) && args(address);
//Pointcut picking out the joinpoint and all the variables
private pointcut MyPointcut(String s, User user, EmailAddress address) :
        cflow(DoSomething(s)) && NewUser(user, address);
```

It is not possible to get a reference to the variable if there is no joinpoint in the cflow has accessed the according variables before.

```
public void DoSomething(Sting s){
        //do something
        EmailAddress address = new EmailAddress(s);
        User user = new User(); //This is the joinpoint we want to trap
}
```

If the code in COTS component is as above it is not possible to get a reference to s, address, and user together if we use AOP to build the glue-code. The reason for this is that there is no joinpoint where all variables are used (or is in the cflow of a joinpoint where the others are used). The solution in this case is to rewrite the code in the COTS component. However, that might not be desirable or even possible in COTS component-based development.

4. A PROCWSS TO COMBINE AOM AND AOP

Most current implementations of aspect-oriented programming start directly from programming level as we did in the case study. First, aspects are identified either by source code reading or document reading. Second, the system will be implemented in the code level based on the current defined aspects. As we have learned from the case study, the risk of implementing a system this way is that many AOP benefits will lose without a good aspect-oriented design.

AOM techniques allow system developers to address crosscutting requirements during system design. The design models consist of a set of aspects and a primary model. The aspect models and primary models can be weaved together by AOM weaving. It is therefore possible to test the validity of a particular aspect model and do trade-off analysis between different models.

To meet the limitation of AOP we combine AOP and AOM to utilize the strength of both approaches. Figure 3 illustrates the combined AOM and AOP approach, where AOM handles the requirement and design phase of the development and AOP handles the implementation phase.

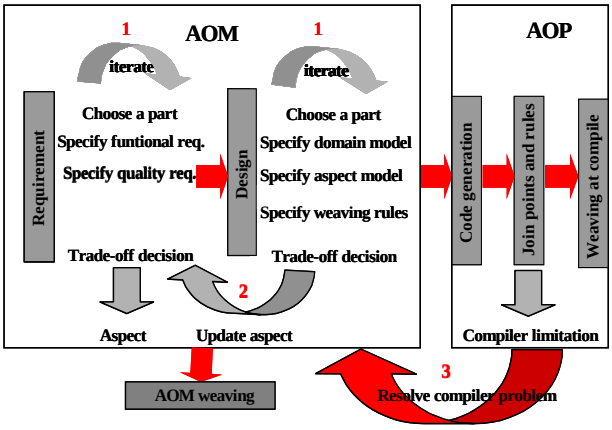

Figure 3. Overview of the combined AOM and AOP development process

- In the requirement phase AOM is used to extract aspects and do trade-off analysis between different aspects.

  When developing security critical and fault tolerant systems non-functional requirements are of great importance. Security and fault tolerance issues may be regarded as crosscutting concerns and treated as aspects. Furthermore, security and fault tolerance issues may also be conflicting and one needs to make trade-off between the two as early as possible in the development as possible. For example, splitting services on two machines increases fault tolerance, but decrease the level of security since one then needs to secure two machines instead of one and in addition, the communication between the two machines. The specification of aspects for security requirements can be done using e.g. UMLsec [10, 11]. UMLsec makes it possible to do verification of the fulfilment of these requirements at later stages in the development.

- In the design phase, AOM is used to describe the system design. A primary model and a set of aspect models may be defined using AOM. In this phase we do conflict testing, functional testing, and verification of fulfilment of requirement specification (both functional and non-functional requirements). Conflict testing is done using AOM weaving. An aspect model must be instantiated before it can be composed with a primary model. The instantiated forms of aspect models are referred as *context-specific* aspect. The instantiation of an aspect model is determined by mapping rules, where a mapping rule specifies the points where a primary model at which aspect elements will be incorporated.

  AOM is used in the requirement and design phase to ensure a proper aspect-oriented design. It can test any conflicting situations and enhance trade-off between aspects. Performing proper conflicting and trade-off analysis in requirement and design phase is more cost-effective than in the coding phase.

- In the coding phase, AOP is used to implement the primary model and aspect models separately. The final system will be weaved together by AOP tools on the code level.

  Some AOM researchers propose that the aspects and the primary model are weaved together in such a way that the aspect is integrated into the primary model before code generation or coding in general. One may then use any code generation approach or manually implementation approach to realize the system once the weaving or composition is done. However, one major problem of this solution is that the separation of the aspects and the primary model is lost once the weaving is done. In addition, the only backtracking possibility once the composition is done is to un-weave using the mapping rules backwards. Since this is not linked to the coding level, changes in the code will not be reflected in the structure of the models and cannot be un-weaved at the modelling level.

  In our process, we propose to keep the separation until the code phase. AOM weaving is only used to test the validity of aspect models. In this way, we can keep the primary model and aspect models separately in the coding phase. It is then easy to backtrack if there is any changes in the code since the generated primary code can be mapped directly back to the primary and aspect models.

Although the process is sequential in general, there are several possible iterations. Arrow number ***1*** in Figure 3 illustrates the fact that each phase may iterate with itself a number of times before moving on to the next phase. In the requirement phase, new non-functional crosscutting requirements may be discovered.

When aspects are identified in the design phase the process iterates back to the requirement specification phase, as illustrated by arrow number ***2*** in Figure 3, and the non-functional requirements are updated. This depends on the system in question and whether the development is done incremental or not. By doing so one can attend to conflicting aspects, situations where you can have one non-functional requirement fulfilled, but not both.

We do not put any limitations on how aspects are defined and described in the AOM part of the development. To meet the limitations of AOP and AOP tools (see section 3.2.2), which varies from tool to tool, the process iterate back to the AOM part of the development whenever limitations of the AOP tool make the design in AOM can not be implemented. This iteration is illustrated by arrow number ***3*** in Figure 3. After the limitation of AOP tools have been discovered, the actions need to be done are:

- The aspects affected by the changes must by located
- The aspect initiation rules of aspect must be checked or revised to make a new *context-specific* aspect. For example, some new AspectJ versions [19] support intertyping and it is therefore possible to intertype aspects into primary model. However, prevsiou AspectJ versions do not support this functionality.
- The aspect model and primary model need to be composed again to check the validity and do trade-off analysis.

In the next section we will illustrate the combined AOM and AOP approach by giving a small example.

5. EXAMPLE

The example used is an e-commerce system where the primary service is selling books online. The system should be able to distinguish between different users and to provide a secure log on and payment service. In order to provide secure log on and transfer of data we need to either encrypt information on the application level or encrypt the link used for transfer. We can use different types of encryption algorithm and techniques, such as private and public key encryption techniques or the DES or Blowfish encryption algorithm. Such issues are crosscutting since encryption may be used by more than one module in the system. In this case one use encryption during log on and payment.

In the following we will only present some of the functional and security requirements for the e-commerce system. The reader is referred to [13] for more information. The relevant functional requirements from the requirement specification in [13] are:

- Consumer has to register sufficient information for contact and identification. The following information has to be entered; *name, password, user-name, email address, address of residency, zip code* and *country*.
- Using registered user-name and appurtenant password, a consumer can log on to the system.
- When services are ordered the consumer pay by giving credit card number and expiration date. This information is subsequently used by the supplier to credit the card when paying for services. The card is credited upon delivery of the service.

The relevant security requirements from the requirement specification in [13] are:

- *Confidentiality* of communication must be ensured in transactions between consumer and the system. Since the communication is through the Internet it has to be encrypted to prevent other parties from being able to read the content of the messages between consumer and system. User name, password and credit card information must be protected by encryption to ensure the secrecy of the system and user. Other information exchanged must be encrypted to ensure privacy of the user.
- *Authenticity* must be ensured to avoid attackers posing as registered consumers. Weak authentication will suffice. This is done for simplicity for the users trying to utilize the system. User must have a unique user name and password for authentication. The password must be at least 8 characters long and contain uppercase and lower case letters as well as at least one number.

We have two crosscutting aspects namely the security requirements for *confidentiality* and *authenticity*. These two aspects both crosscut all three functional requirements.

5.1 Using AOM in the requirement and design phase

In the requirement and design phase, AOM is used to ensure a valid and efficient design of the whole system.

5.1.1 Primary model

We use activity diagrams to describe the functional requirement of the system as depicted in Figure 4. Figure 4 describes *partly* (To describe a primary model completely, other diagrams, such as class diagram, state diagrams, are also needed) the primary model of the system. The crosscutting aspects, i.e. confidentiality and authenticity, will be used in the sub-parts that are shadowed in this model.

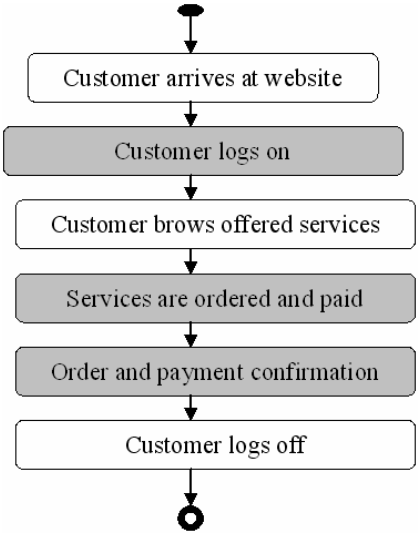

Figure 4. The primary model of the system

5.1.2 Aspect model

The main point with aspects is not only that they are crosscutting, but also increase of the level of reusability and ease of software evolution. For instance, if we implement a weak authentication mechanism and later decide that we need to update and strengthen the mechanism, we only need to update the aspect as long as the interface between the aspect and the primary model remains the same.

In this example we illustrate the aspects for confidentiality and authenticity using simplified protocols. To address the security requirement for confidentiality of information we use secrete-key encryption. In this example we do not specify the cryptographic algorithm used, since this is transparent and handled by the aspect. Figure 5 depicted an activity diagrams that describe a simplified confidential requirement of the system. The diagram describes *partly* (To describe an aspect model completely, other diagrams, such as class diagram, state diagrams are also needed) the confidentiality aspect model of the system. To initiate a *context-specific* aspect, pointcuts must be defined. We define that the confidentiality activity should happen "after" the functions of the payment and payment confirmation, and "after" the function of customer logs on.

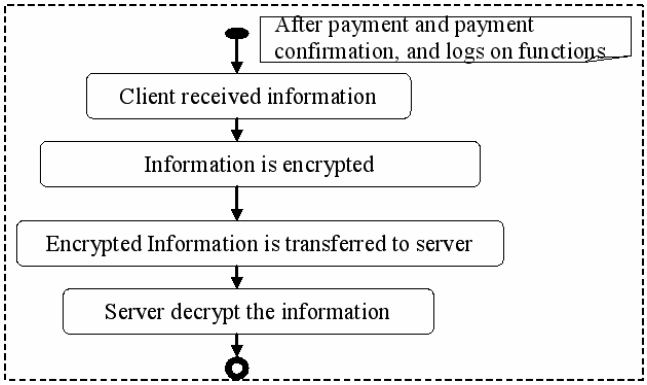

Figure 5. The confidentiality aspect model

We also use activity diagrams to describe the authentication requirement of the system as in Figure 6. The pointcut for this aspect is "after" the function of customers logs on.

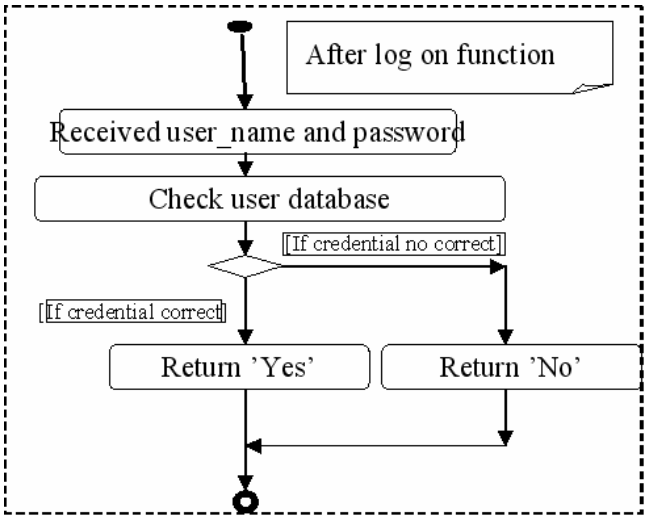

Figure 6. The authentication aspect model

5.1.3 Weaved model for primary model and authentication aspect

We now have the primary model and the aspects models. According to the process these two models should now be composed in order to validate the fulfilment of the requirements and to reveal any logic problem in the model. The verification of the fulfilment can e.g. be done using the UML extension for secure systems development, UMLsec [10, 11] and the verification and validation tool for UMLsec [9]. ***The weaving on model level is simply done to reveal any problems before transforming the models to AOP***.

In this example we only demonstrate a manual weaving on the model level for the authentication aspect. There exist several weaving techniques, both manual and automated. One can compose models using matching of names, syntactic compositions or element properties expressed in OCL, semantic composition. In this case we have used a manual semantic composition strategy. The weaved model is shown in Figure 7.

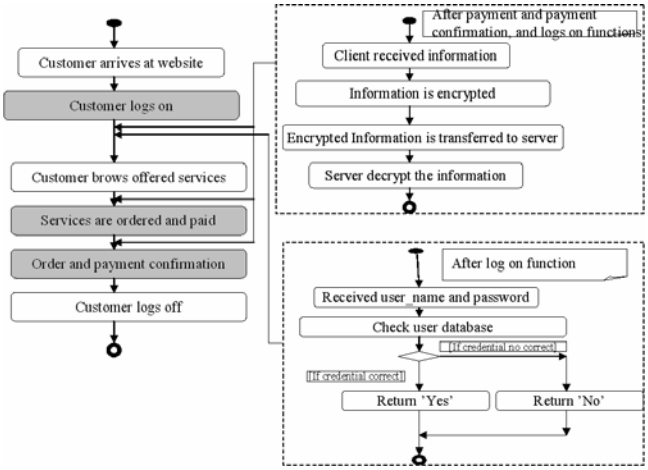

Figure 7. The weaved model

5.2 Using AOP in the implement phase

As we emphasized above, we use AOM in the requirement and design phase to ensure a valid and efficient design. However, we will not implement the system based on the AOM-weaved model. The primary model and the aspect models will be implemented separately by using AOP tools. The actual weaving is done on the coding level using AOP weaving.

5.3 Possible iterations

There are several possible iterations in our proposed process.

- Each phase, i.e. requirement phase, design phase may iterate with itself a number of times before moving on to the next phase. In the requirement phase, new non-functional crosscutting requirements, such as performance requirements, may be discovered. In such cases a new performance aspect model is therefore needed.
- When aspects conflicts are identified in the design phase the process iterates back to the requirement specification phase. For example, too complexes encrypt algorithm may make the system very slow, which conflict with the performance tuning aspect. A trade-off balance in requirements is therefore needed.
- The actual weaving is done on compiler level using AOP. However, there are limitations in AOP supported compilers as we showed in the case study. If such situations appear we iterate back to the design level in order to update the model before proceeding.

6. CONCLUSION AND FURTHER WORK

In this paper, we present a combined AOM and AOP approach. The motivation of the study comes from lessons learned in a case study, i.e., a good aspect-oriented design is essential to get the benefit of AOP and limitations in AOP tool may require a revise of the design.

In our approach, AOM is used in the requirement and design phase while AOP is used in the implementation phase. Aspects are defined in the requirement phase. In the design phase, AOM is used to describe the system design. Conflicting and trade-off analysis are performed in both the requirement phase and the design phase using AOM weaving. In the coding phase, the AOM primary model and aspect models are coded using AOP tools separately. The separately coded primary model and aspect models are weaved together using AOP weaving afterwards.

The approach may require several iterations of each phase before moving to the next phase. Aspects may need to be revised if new non-functional requirements are discovered. The requirement and design need to be changed if new aspects are identified in the design phase. In the coding phase, the limitations of available AOP tools may require changes in system design.

To improve our approach in practice, several future studies needs to be done:

- An extended Aspect-UML is needed to express aspect model. Pointcust, jointpoints and advices etc. cannot be expressed exactly using current UML tools. As aspects are context-specific and need initiation, Aspect-UML need to be able to customize the aspect model based on different contexts. Another requirement for Aspect-UML is that the connections between aspects and classes should be detailed enough to make the tracking easy. This requires that both the advices and pointcuts are represented in the aspect model and a line shows the exact point(s) each pointcut picks out.
- A composition method to weave primary model and aspect model is required. As most aspects will be integrated into primary code based on the runtime information or current state, the composition method should reflect this dynamic character.

7. REFERENCES

[1] Clarke, S., Harrison, W., Ossher, H., and Tarr, P., Separating concerns throughout the development lifecycle. In *Proceedings of the 3rd ECOOP Aspect-Oriented Programming Workshop* (Lisbon, Portugal, June 14-18, 1999). Springer Lecture Notes in Computer Science, Vol. 1743, 299.

[2] Colyer, A., and Clement, A., Large-scale AOSD for Middleware. In *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, (Lancaster, UK, March 2004), ACM Press, New York, NY, 2004,56-65.

[3] Fiadeiro, J. L. and Lopes, A., Algebraic semantics of co-ordination or what is it in a signature? In *Proceedings of the 7th International Conference on Algebraic Methodology and Software Technology (AMAST'98)*, (Amazonia, Brasil, Jan 1999) Springer Lecture Notes in Computer Science, Vol. 1548, 293-307.

[4] Gray, J., Bapty, T., Neema, S., and Tuck, J., Handling crosscutting constraints in domain-specific modelling. *Communications of the ACM*, 44, 10 (Oct 2002), 87-93.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, Aspect Oriented Programming. In *Proceedings of 11th European Conference on Object-Oriented Programming* (Jyväskylä, Finland, June 9-13 ,1997), Springer Lecture Notes in Computer Science, Vol. 1241, 220-242.

[6] Georg, G., France, R., UML Aspect Specification Using Role Models. In *Proceedings of the 8th International*

Conference on Object-Oriented. Information Systems (OOIS 2002 (Montpellier, France, September 2-5, 2002), Springer Lecture Notes in Computer Science, Vol. 2425, 186-191.

[7] Georg, G., France, R., and Ray, I., Designing High Integrity Systems using Aspects. In *Proceedings of the 5th IFIP TC-11 WG 11.5 Working Conference on Integrity and Internal Control in Information Systems (IICIS 2002)* (Bonn, Germany, Nov 2002), Kluwer Academic Publishers, Norwell, MA, USA, 37-57.

[8] Georg, G., and France, R., and Ray, I., An Aspect-Based Approach to Modeling Security Concerns. In *Proceedings of the Workshop on Critical Systems Development with UML* (Dresden, Germany, September 30, 2002), 107-120.

[9] Houmb, S.H., Jürjens, J. Developing Secure Networked Web-based Systems Using Model-based Risk Assessment and UMLsec. In *Proceedings of 10th Asia-Pacific Software Engineering Conference (APSEC 2003)* (Chiang Mai, THAILAND, December 10-12, 2003), IEEE Press, 2003, 488-499.

[10] Jürjens, J. *Principles for secure systems design*. PhD thesis, Wolfson College, 2002.

[11] Jürjens, J. UMLsec: Extending UML for secure systems development. In *Proceedings of the 5th International Conference on Unified Modeling Language (UML 2002)* (Dresden, Germany, September 30 - October 4, 2002), Springer Lecture Notes in Computer Science, Vol. 2460, 412–425.

[12] Li, J., Bjørnson, F.O., Conradi, R., and By Kampenes, V. An Empirical Study of COTS Component Selection Processes in Norwegian IT companies. In *Proceedings of International workshop on models and processes for the evaluation of COTS components* (Edinburgh, Scotland, May 2004), IEE ISBN-0-86341-422-2, 27-30.

[13] Lillevik, Ø. *A model-based approach to handling risks in security critical systems*. Master Thesis NTNU, http://www.stud.ntnu.no/~lillevik/CORAS/masterthesis.pdf, 2003.

[14] Rashid, A., Sawyer, P., Moreira, A., and Araujo, J., Early Aspects: A Model for Aspect-Oriented Requirements Engineering. *IEEE Joint International Conference on Requirements Engineering*. IEEE Computer Society Press, pp 199-202, Essen, Germany, Sept 2002.

[15] Suzuki, J. and Yamamoto, Y., Extending UML with Aspects: Aspect Support in the Design Phase. In *Proceedings of the 3rd ECOOP Aspect-Oriented Programming Workshop*, (Lisbon, Portugal, June 14-18, 1999). Springer Lecture Notes in Computer Science, Vol. 1743, 299-300.

[16] Tzilla Elrad, Mehmet Aksits, Gregor Kiczales, Karl Lieberherr, and Harold Ossher, Discussing Aspects of AOP. *Communications of the ACM*, 44, 10 (October 2001), 33 -38.

[17] The Object Management Group. *The Unified Modeling Language*. OMG, formal/2003-03-61, version 1.5, 2003.

[18] Java Email Server: Getting started, available at: http://www.ericdaugherty.com/java/mailserver/gettingstarted .html (this reference need more information)

[19] AspectJ, available at: http://eclipse.org/aspectj/