

A Case Study on Improving Changeability of COTS-Based System Using Aspect-Oriented Programming*

JINGYUE LI¹, AXEL ANDERS KVALE¹ AND REIDAR CONRADI^{1,2}

¹*Department of Computer and Information Science*

Norwegian University of Science and Technology

No-7491 Trondheim, Norway

E-mail: jingyue@idi.ntnu.no

²*Simula Research Laboratory*

P.O. Box 134, No-1325 Lysaker, Norway

More and more software projects are using COTS (commercial-off-the-shelf) components. One of the most challenging problems in COTS-based development is to evolve a system to follow changes in the customer requirements. It is therefore important to increase the changeability of the COTS-based system, so that new component can easily replace the problematic COTS components. Aspect-Oriented Programming (AOP) claims to make it easier to develop and maintain certain kinds of application. We performed a case study to investigate whether AOP can help to build an easy-to-change COTS-based system. This compared the changes when adding and replacing components between a COTS system implemented using Object-Oriented Programming (OOP) and the same system implemented using AOP. The results show that integrating COTS components using AOP may help to increase the changeability of the COTS-based system if the cross-cutting concerns in the code used to glue the COTS component are homogeneous (i.e., have a consistent application of the same or very similar policy in multiple places). Extracting heterogeneous or partially homogeneous cross-cutting concerns in glue-code as aspects does not give any benefits. This study also discovered some limitations in the AOP tool that makes it difficult or even impossible to integrate COTS components with AOP.

Keywords: object-oriented programming (OOP), aspect-oriented programming (AOP), commercial-off-the-shelf (COTS)-based development, component-based software engineering (CBSE), changeability

1. INTRODUCTION

As the use of COTS promises faster time-to-market and increased productivity [14], COTS-based development has become increasingly important in software and system development. At the same time, COTS-based development introduces many risks. An empirical study performed by Li *et al.* investigated the experience of COTS-based projects in three European countries and ranked some typical risks (problems) in COTS-based projects [14]. The study discovered that one main challenge in COTS-based development is to follow changes in the customer requirements. Customer requirements

Received July 1, 2005; accepted November 24, 2005.

Communicated by Sung Shin.

* The preliminary version of this paper was presented at the Software Engineering Track of the 20th Annual ACM Symposium on Applied Computing (ACM SAC 2005), Santa Fe, New Mexico, March 13-17, 2005.

usually change faster than COTS components. Due to the “black-box” property of COTS components, the COTS integrator cannot change the source code to satisfy their customers’ changed requirements.

It is necessary to prepare for adding new COTS components and replacing currently unsuitable COTS components to manage these risks. In the process of adding and replacing the COTS components, some component-relevant codes (i.e. glue-code) may need to be written or rewritten. Although glue-code development usually accounts for less than half the total COTS-based software development effort, the effort per line of glue-code averages about three times the effort per line of development application code [25]. It is therefore important to decrease the necessary changes of the glue-code to save the cost of maintaining and evolving a COTS component-based system.

Aspect-Oriented Programming (AOP) is claimed to increase the maintainability of a system compared to Object-Oriented Programming (OOP) [5]. In COTS-based development, glue-code that invokes the COTS component functionalities may be scattered all over the system. If scattered cross-cutting concerns in the glue-code can be separated into aspects, it will be easier to understand and change the system. A case study was done to empirically investigate how to build an easy-to-change COTS-based system using AOP. This compared the needed changes in glue-code during the process of adding and replacing COTS components. The amount of classes changed and the amount of lines-of-code (LOC) changed during adding and replacing COTS components are compared between an aspect-oriented system and the object-oriented version.

Results from this study confirm the main benefit of using AOP, that is, all the changes are centralized in the aspect without scattering everywhere in the whole system [5]. However, the lessons we learned show that detailed plans and designs should be performed before the decision is made to use AOP in COTS-based development. The AOP users should ensure that the cross-cutting concerns in the glue-code are homogeneous. The AOP user must also be aware that the limitation of current the AOP tool may not provide full support to integrate COTS component.

The rest of this paper is organized as follows: Section 2 introduces some related concepts and background to this study. Section 3 describes our case study design. Section 4 presents the implementation of the case study and results. Section 5 illustrates the three main lessons learned. Section 6 discusses the results of this study and compares this study with related work. The conclusions and future works are presented in section 7.

2. BACKGROUND

2.1 COTS Component Definition

A component is a unit of composition, and must be specified so that it can be composed with other components and integrated into a system (product) in a predictable way [8]. That means that a component is an “executable unit of independent production, acquisition, and deployment that can be composed into a functioning system.” For a COTS component, we have used the definition by Torchiano and Morisio [16], where a COTS component is a software element that:

- Is either provided by some other organizations in the same company, or provided by external companies as a commercial product.
- Is integrated into the final delivered system.
- Is not a commodity, i.e. not shipped with an operating system, not provided with the development environment, and not generally included in any pre-existing platforms.
- Is not controllable by the user, in terms of provided features and their evolution. Our addition: This normally means “*black box*”, i.e. no source code is available.

This definition means that we include not only components following COM, CORBA, and EJB standards, but also C++ or Java™ libraries. This definition is consistent with the scope in the component marketplace [4].

2.2 Build and Rebuild Glue-Code in a COTS-Based System

A COTS component-based process consists of several phases, comprising: COTS component assessment and selection; COTS component tailoring and integration; maintenance of COTS and non-COTS parts of the system [3]. In each phase, the glue-code will be built or rebuilt.

Several formal selection processes and decision making methods have been proposed to select and evaluate COTS components, [21]. In some of these proposed selection processes, the hands-on trial is regarded as a necessary step [12, 17]. The hands-on trial means building glue-code and integrating the COTS components into possible future environment in order to test their quality and compatibility with other components in the system. Selecting a proper COTS component from several possible candidates by integrating and testing them is time-consuming, especially if the glue-code spread throughout the system. A lot of glue-code needs to be modified and rewritten to change the testing from one COTS component to another.

When integrating COTS components, the internal implementation may cause mismatches between these COTS components. Therefore, glue-code may be needed to integrate these COTS components and make them work together.

In the maintenance phase of a COTS-based system, one possible risk is that the vendor may go bankrupt and fail to support the current COTS component running in the system. Some vendors may withdraw support for the old version component when they publish the new version. The new component version may have no backwards compatibility with the old versions. Another possible risk in this phase is that current COTS components cannot satisfy the changed requirements. In this case, the glue-code might be modified to integrate the new or replaced COTS components [14].

2.3 Aspect-Oriented Programming

Aspect-oriented programming (AOP) is a new programming paradigm that takes another step towards increasing the design concerns that can be captured clearly within source code [5]. An aspect (see Appendix for definition) is a modular unit of cross-cutting implementation, and it encapsulates behavior that affects multiple classes. With AOP, each aspect can be expressed in a separate and natural form, and can then be automatically combined into a final executable form by an aspect weaver (see Appendix for

definition). As a result, a single aspect can contribute to the implementation of a number of procedures, modules, or objects. It therefore helps to increase reusability of the source code [5]. AOP brings several new ideas and definitions. Brief descriptions of AOP terminologies used in this study are listed in the Appendix. Detailed definition of each terminology can be found at [27, 28]. Several different AOP tools have been built, such as AspectJ [6], AspectWerkz [2], and JBoss AOP [11].

3. DESIGN OF THE CASE STUDY

Most current glue-code is built using OOP. The advantage of using AOP over OOP is that it is possible to modularize code that cross-cuts the whole application. In COTS-based development, the use of COTS component functionalities or methods are scattered in multiple classes of the system. If the cross-cutting concerns in the code to glue the COTS components can be separated into aspects, it will probably be easier to change the system. Most previous empirical studies on AOP focused on components that can be modified completely [15, 20]. However, there were few studies on integrating the COTS components (where the source code is either not available or hard to modify) using AOP [19].

In this study, we use the GQM (Goal-Question-Metrics) approach [26] to investigate whether AOP can help to increase the changeability of the COTS-based system. The *goal* of this study is to compare the changeability of a COTS-based system implemented using AOP with the changeability of a COTS-based system implemented using OOP. To reach this goal, we ask the following two research questions **RQ1** and **RQ2**:

RQ1: Is it cheaper to add a new COTS component in a COTS-based system using AOP than using OOP?

As we mentioned in section 2.2, glue-code is often needed to either integrate the COTS components initially or add COTS components afterwards. The research question **RQ1** compares how many changes are needed to integrate a COTS component by AOP vs. OOP.

To investigate **RQ1**, the metrics are the amount of classes changed (added/deleted/rewritten) and the LOC changed in the glue-code while adding a COTS component. Although most studies use person-hours to measure the effort to integrate a COTS component, we do not use this here. The key validity threat of using person-hours is: a developer may have different levels of experience with OOP and AOP. OOP has been used for many years and developers have a lot of experience with it. On the other hand, AOP is a method that developers have only limited experience with.

RQ2: Is it cheaper to replace a COTS component in a COTS-based system using AOP than using OOP?

As we indicated in section 2.2, the glue-code needs to be rewritten to response to replacing the problematic components. The research question **RQ2** compares how many changes are needed to replace one COTS component by another.

To investigate **RQ2**, the metrics are the amount of classes changed and the LOC changed in the glue-code during the process of replacing COTS components.

4. IMPLEMENTATION AND RESULTS

There are two possible strategies to implement this study:

- Re-engineer an existing object-oriented system using AOP.
- Build two systems from scratch, one using OOP and another using AOP.

Although some previous studies chose to build two systems from scratch [20], we decided to re-engineer an existing system because:

- It will be easier to compare the results since two systems are identical, except that the cross-cutting concerns in the glue-code are extracted into aspects in the AOP implementation.
- It can avoid the possible bias of building two separate systems (one using OOP and the other using AOP) from scratch. If we built two systems from scratch, the results might be decided by system design and implementation (i.e. a specific problem is solved elegantly in the OOP implementation and poorly in the AOP implementation), and not by the basic difference between AOP and OOP.

4.1 System and Programming Language Selection

The key characteristic of AOP is that the aspect code is combined with the primary programming code by an aspect weaver. In our study, the AOP tool should be able to do byte-code weaving because we focus on COTS components, which are supposed to be delivered in byte-code format. Therefore, the selected aspect-oriented tool is AspectJ version 1.1 [6], which can weave an existing byte-code COTS component without the source code.

The system chosen for the study is an open source Java Email Server (JES), the JES server [9]. It is built by OOP principles using Java™. There are 21 classes and about 1400 LOC (excluding code in the libraries) in the JES server.

In this study, we treated some Java™ libraries as components. We treat this open source JES as a COTS-based system by *not reading and changing the source code inside components, even if the source code is available. Thus we only study glue-code, not the source code inside a component.*

4.2 Case Study Steps

We designed three steps in this study. The **first step** was to re-engineer the JES server by extracting cross-cutting concerns into aspects with AspectJ. We call the original JES server the OOP JES and the re-engineered JES server the AOP JES.

In the OOP JES server, we found several cross-cutting concerns, such as exception handling, logging, and shutdown handling. The logging was implemented by using a separate library *log4j*, which can be regarded as a COTS component. The *log4j* is used by several classes. We therefore moved the glue-code that is relevant to *log4j* into a separate aspect using AspectJ (Version 1.1).

The **second step** was to investigate **RQ1**, that is, the effort of adding a component using AOP vs. using OOP. In this step, a popular spam-checker for most email servers (*SpamAssassin* [22]) was added to the JES server.

The JES server uses the class *SMTPMessage* to keep the email and functions related to an email message. In the OOP JES, spam-checking with *SpamAssassin* is created inside the *SMTPMessage* class. When the *SMTPProcessor* class has received a new email and stored it inside a *SMTPMessage*, the spam-checking will be called and a Boolean value will be returned as the result to indicate whether the email is a junk mail.

In the AOP JES, the *SpamAssassin* is implemented inside an aspect. The procedure of checking a *SMTPMessage* is the same as in the OOP JES. The difference is that the spam-checking is called inside the aspect. A pointcut picks out all the places where a spam-mail should be checked for spam. The aspect calls *SpamAssassin* and returns the Boolean result. In the AOP JES, every *SMTPMessage* that is created by *SMTPProcessor* is checked by *SpamAssassin* without knowing the existence of *SpamAssassin*, because all spam-checkings are done inside an aspect.

In order to add the spam-checking component *SpamAssassin*, the total amount of classes and LOC changed in the OOP and AOP JES are shown in the following Table 1.

Table 1. Changes performed to add SpamAssassin.

Changes	OOP JES	AOP JES
Amount of classes changed	2 classes	1 aspect
Amount of LOC changed	36	44

Since only two classes (*SMTPMessage* and *SMTPProcessor*) were changed to add the *SpamAssassin* in the OOP JES, more LOC were added in the AOP JES than the OOP JES. This is because AOP JES needed extra code lines to define pointcuts. However, we should interpret this result carefully. If several classes invoke the *SpamAssassin*, less LOC might be added in the AOP JES, since only a new pointcut definition is needed for each additional use. In the OOP JES, all classes that use spam-checking functions need to add relevant glue-code. So, the answer to **RQ1** is: adding a COTS component using AOP is not absolutely cheaper than using OOP. When adding a COTS component, fewer classes need to be changed in system using AOP than in one using OOP. However, the possibility of less LOC needs to be changed in system using AOP depends on how many classes use functions of the COTS component and the complexity of the pointcuts.

The **third step** was to investigate **RQ2**, that is, the effort of replacing a COTS component. As mentioned in the first step, we rewrote the glue-code relevant to *log4j* using AOP. In this step, we used another logging component to replace the *log4j* component in both the OOP and AOP JES. In the JavaTM Development Kit (SDK) version 1.4, there is a new logging component available, named *util.logging* [10]. It built up in the same way as *log4j*. We therefore decided to replace the *log4j* component with the *util.logging* component. There are three steps in both the OOP and AOP JES:

- The first change was the initialization of the logging system. In the original version

with *log4j*, this was done inside the system before the first log-object can be created. With the *util.logging*, the initialization is done through an XML-file. The XML-file is passed to the system by the Java™ command to start the system.

- The second change was the declaration and initialization of the log component. To implement this change, all declarations must be changed to use *util.logging*, and all initializations of the objects must use the new syntax of *util.logging*.
- The third change was the way a message is written to the log. *Log4j* and *util.logging* use slightly different syntax when appending a log-message. The *log4j* uses syntax like *log.warn* and *log.info*. The *util.logging* requires a level to be supplied to every message with the form *log.log (Level.LEVEL, String)*.

In the process of replacing the *log4j* component with *util.logging*, the total amount of classes and LOC changed in the OOP and AOP JES are shown in Table 2.

Table 2. Changes performed to replace log4j by util.logging.

Changes	OOP JES	AOP JES
Amount of classes changed	12 classes	1 aspect
Amount of LOC changed	184	162 (In aspect)

In the **third step**, we also used another spam-checking component (*SpamProbe* [23]) to replace the *SpamAssassin* component in both the OOP and AOP JES. *SpamProbe* is a spam detection application using Bayesian analysis of terms contained in the email.

- In the OOP JES, the *SpamAssassin* was changed to *SpamProbe* in the SMTPMessage class. The returned result is a Boolean value to indicate whether the email is a junk mail.
- In the AOP JES, the change is the same as in the OOP JES. However, only the aspect is changed.

In the process of changing the *SpamAssassin* component with *SpamProbe*, the total amount of classes and LOC changed in the OOP and AOP JES are shown in the following Table 3.

Table 3. Changes performed to replace SpamAssassin with SpamProbe.

Changes	OOP JES	AOP JES
Amount of classes changed	1 class	1 aspect
Amount of LOC changed	15	15

Tables 2 and 3 allow us to summarize the result of **RQ2**: it is not absolutely cheaper to replace a COTS component in a COTS-based system using AOP than using OOP. When replacing a COTS component, fewer classes need to be changed in the system us-

ing AOP than in the system using OOP. However, AOP does not ensure that less LOC need to be changed when replacing a COTS component. Detailed analysis of this result is given in sections 5 and 6.

5. LESSONS LEARNED

Although we performed only a limited empirical investigation, the lessons learned from this study illustrate some important insights in using AOP in a COTS-based system.

5.1 The First Lesson Learned: Extracting Partly Homogeneous Cross-Cutting Concerns in Glue-code into Aspects Does Not Bring Benefit

Logging functions are usually referred to as the “typical” ideal example to be extracted into aspects [1]. The place where logging is needed can be trapped by a joinpoint. A logging aspect can then easily log all the run-time information available at every joinpoint. This makes it easy to implement logging into several classes with only the minimum amount of LOC and without changing any of the original classes.

In this study, we also extracted glue-code relevant to logging into aspects. However, data in Table 2 show that the amount of LOC changed in OOP JES and AOP JES is almost the same, when replacing the *log4j* component. The reason is that *log4j* glue-code that is spread in the system is not homogeneous (consistent application of the same or very similar policy in multiple places). In this study, the heterogeneity comes from the static strings to be printed out in the OOP JES, as shown in Code 1.

```
// print out logging information after the change of X and Y
li.log.info("Changed X and Y to," ...)
// print out logging information after the change of string
li.log.info("Changed String," ...)
```

Code 1. Code for logging in the OOP JES.

Printing out the above static strings to give a trace of the actual exception is common in OOP systems. However, these static strings bring difficulties to the AOP implementation. The logging information acquired in the AOP system is limited to the information provided by the joinpoint (name of the function, name of the enclosing function, arguments, class name, and so on). To output the same information as the OOP JES, we had to build several quite complex pointcuts to define what we want to log, as shown in Code 2. We therefore partly lost the benefits (and strengths) of using AOP.

5.2 The Second Lesson Learned: AspectJ does not Provide Complete Support to Access Variables in the Program

When we re-engineered the AOP JES, we learned that some unexpected limitations of AspectJ (Version 1.1) made it impossible to use AOP in the COTS-based development. The reason is that AspectJ 1.1 requires changing the source code of the COTS compo-

ment in some cases. However, changing source code is not generally available in COTS-based development. The details are as follows:

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

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

If other variables than the ones mentioned above are needed, several pointcuts are needed to get references to these variables. If we want to access the input string *s* when the *user* is created in the sample Code 3, we need to combine several pointcuts as shown in Code 4.

```
// 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 #2
after(LogInterface li, String s) returning: PC1(li, s) {
li.log.info("Changed name to" + s); }
```

Code 2. Code for logging in the AOP JES.

```
public void DoSomething(String s) {
    EmailAdress address = new EmailAddress(s);
    User user = new User(address); // The joinpoint we want to trap
}
```

Code 3. Sample code to create a user with email address.

If there is no joinpoint in the **cflow** that has accessed the relevant variables before, AspectJ version 1.1 does not support getting a reference to these variables.

For example, if we use AOP to build the glue-code as shown in Code 5, it is not possible to get a reference to *s*, *address* and *user* together. The reason is that no joinpoint (or cflow) used all these three variables at the same time.

The solution in this case is to rewrite the source code from the “User ()” to “User (String)”. However, this might not be desirable or even possible in COTS-based development, if we regard the class “User” as part of a COTS component.

```

// Pointcut picking out the extra variable String s
private pointcut DoSomething(String s)
  Execution (void DoSomething(String)) && arge(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)) && arge(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);

```

Code 4. AOP code to access the input string *s* when the user is created.

```

public void DoSomething(String s) {
  EmailAddress address = new EmailAddress(s);
  User user = new User(); // The joinpoint we want to trap
}

```

Code 5. Sample code to create a user without email address.

5.3 The Third Lesson Learned: The Static Typing of Java Can Make the Use of Interfaces in AOP Complex

In AOP, interfaces are often used to get a common handle to several classes of different types. When a cross-cutting concern demands that certain code and functionality must be implemented into several classes, this can often be solved elegantly by creating an interface that contains both the functions and variables the classes need. All one needs to do afterwards is to define the classes that implement this interface. The problem arises when we want to access variables and methods that are not defined in the interface, for example, when all the classes implementing the interface also need a static variable/method. AspectJ (Version 1.1) does not permit static variables/methods to be defined in an interface. So, the static information must be inter-typed into the classes. When picking out which classes to apply an advice to, it is easy to first let all these classes implement an interface and then look for this interface in the joinpoints. The problem is that when we pick out these classes based on this interface, we will get an interface-type reference to the class (we will only “see” the interface variables and methods). In order to access the static variables/methods of the class, we have to cast the reference to the appropriate class type. As JavaTM does not support dynamic typecasting, we cannot decide which class type we want to cast the reference to at run-time. We therefore have to manually test each class and then cast to the appropriate class (type everything statically before we compile the project). An example is shown in the following Code 6. Since the interface is used to pick out all these classes, we have a pointer of type *MyInterface* pointing to an object implementing *MyInterface*. If (usually when) we want to access this object, we need a pointer of type *MyObject*. In JavaTM there is no way of doing this without statically type-casting the object. Since we do not know the type, we have to test to get a reference to the object. These extra codes on testing the type of an object make the aspect implementation complex and prone to error.

```

// defining the pointcut
Private pointcut MyPointcut(MyInterface mi):
    this(mi) && execution(* *(.));
// the advice that we have to typecast
before(MyInterface mi): MyPointcut(mi) {
    if (mi instanceof MyObject) {
        MyObject mo = (MyObject)mi;
        // do stuff
    }
}

```

Code 6. Sample code to test the type of an object before type-casting.

6. DISCUSSIONS

6.1 Comparing AOP with OOP in COTS-based Development

Compared with OOP, our results confirmed some benefits AOP. As we can see in Tables 1, 2, and 3, all the changes were centralized into the aspect. The centralization of changes was claimed to be the main benefit of using AOP [5]. Another AOP benefit is that it can remove the dependencies between the classes using the COTS component and the COTS component itself. All uses of the COTS components can be performed in the aspect. It makes the system oblivious about the existence of the COTS component. However, our results discovered several issues that need to be considered before the decision to use AOP instead of OOP in COTS-based development.

First, the nature of the cross-cutting concerns in glue-code needs to be examined carefully. As we can see in Table 2, the amount of LOC changed in AOP JES was almost the same as the amount of LOC changed in the OOP JES, when replacing the *log4j* component. It illustrates that the benefit of extracting cross-cutting concerns into aspect depends greatly on the nature of these cross-cutting concerns. Extracting partly homogeneous cross-cutting concerns into the aspect does not bring partial benefits.

Second, wrapping the COTS component properly using OOP can also help to centralize the changes when replacing the COTS component. When we added the *SpamAssassin* in the second step, we wrapped the *SpamAssassin* function into the *SMTPMessage* class. All other classes, which need to scan a message, can just call the *SMTPMessage* class to get the result (a Boolean value) without knowing which spam-checking component is used. When we change from *SpamAssassin* to *SpamProbe*, only the *SMTPMessage* class was changed. So, the changes were also centralized into one class, as shown in Table 3.

Third, the limitations of the AOP tool need to be clarified. The second lesson learned shows that the limitation of AspectJ requires changing the source code of a COTS component, which is not expected or possible in most cases. Moreover, most COTS components are delivered as byte-code instead of source code. We therefore need an AOP tool to weave the byte-code. However, available byte-code weaving tools, such as AspectJ and AspectWerkz, support only JavaTM. COTS components expressed in other languages, such as C and C++, cannot be integrated with a byte-code weaver. Since

Java™ was not designed for AOP, the characteristics of Java™ may poorly influence the Java™-based AOP tool, as our third lesson showed.

6.2 Comparison with Related Work

Several previous studies have empirically investigated how to use AOP in different kinds of applications. Walker *et al.* have investigated the claims that AOP is easier to reason about, develop and maintain certain kinds of application code [20]. They compared the efficiency of debugging and changing two systems (one built with AOP and the other built by OOP) with the same functionality. They discovered that the separation provided by AOP seems most helpful when the interface is narrow (i.e., the separation is more complete); while partial separation does not necessarily provide partial benefit. Our results on adding and replacing COTS components give further support to their conclusion. Using AOP makes it is easier to reason about and change a component, if the interface between a COTS component and other parts of the system is completely separated. Other parts of the system are even oblivious about the existence of the COTS component.

Colyer *et al.* investigated AOP by re-engineering a large middleware system [1]. They proposed the challenges and lessons learned in re-factoring both homogeneous and heterogeneous cross-cutting concerns in the middleware. They concluded that it is more challenging to re-engineer heterogeneous cross-cutting concerns than homogeneous ones. Our results on changing a logging component (Section 5.1) provide further support to their conclusion. They proposed processes and methods that can help to change the heterogeneous cross-cutting concerns into ideal aspects. In their system, all source code can be changed. Our study focuses on COTS-based systems. Due to the common “black-box” property of COTS components, it is more difficult to extract heterogeneous cross-cutting concerns into good aspects.

Lippert *et al.* investigated the benefits of AOP by re-engineering an OOP system and extracting exception detection and handling this as aspects [15]. They concluded that AOP provides better support for reuse. Our results give further support to their conclusion. In case the cross-cutting concerns in glue-code of the COTS component are homogeneous, the reusability of the glue-code using AOP will be higher than using OOP, because fewer classes and LOC need to be changed when adding or replacing a component. When the cross-cutting concerns in the glue-code are not or only partially homogeneous, the only benefit of AOP is that fewer classes need to be changed. However, it does not mean that less LOC need to be changed in the AOP implementation than in the OOP implementation.

Other studies have tried to integrate the AOP into a component model, such as CORBA, and .NET [18, 24]. If a COTS component follows these component models, it will be very easy to plug components in and out. However, there are still few COTS components on the market that follow these new component models. Our study is therefore limited to the COTS components that are found on the market, such as components in the form of Java™ libraries.

6.3 Possible Threats to Validity

The threat to *internal validity* of this study is that (some) limitations of AOP tools

may destroy the possible AOP benefits. Although the intention of this study is to compare the benefit of the AOP to OOP approaches, the differences in the maturity of AOP and OOP tools may affect the result of this study.

The threat to *construct validity* is that we used the amount of classes and LOC that were changed to measure the changeability of the system. There are several metrics proposed to measure the changeability of OOP systems. However, few studies have proposed well-defined and tested metrics to measure changeability in AOP. Walker *et al.* used the time needed to debug and change a system as the metrics [20]. However, the value of this metrics depends on respondents' experience with AOP and OOP. We therefore selected a more objective metrics, i.e. amount of classes and LOC that were changed.

The possible threat to the *external validity* of this study is that the size of our system is rather small. We only did a limited empirical investigation of some fine-grained COTS components. However, the study discovered some important issues when using AOP in COTS-based development.

Concerning *conclusion validity*, this is a pre-study for our further investigations. The intention of the present study is to draw out ideas that may be transferred to other cases.

7. CONCLUSIONS AND FUTURE WORKS

We have studied how AOP eases the adding and replacement of components in COTS-based development. We re-engineered an existing OOP application using AOP and compared the amount of classes and LOC of glue-code changed during adding and replacing COTS components. From this study, we found that:

- When adding or replacing a COTS component, the main benefit of using AOP in a COTS-based system is that fewer classes need to be changed than using OOP. However, using AOP does not ensure that less LOC need to be modified when adding or replacing COTS components. It depends on whether the cross-cutting concerns in the glue-code are homogeneous. Using AOP when cross-cutting concerns are (partly) heterogeneous may not bring benefits. In addition, proper wrapping of COTS components using OOP can also get the same benefit from improving the changeability of the COTS-based system. A careful analysis of cross-cutting concerns in the glue-code and the nature of the COTS component is therefore needed, before the decision is made to use AOP in COTS-based development rather than OOP.
- To integrate COTS components using AOP, the AOP tools need to be investigated in detail, because limitations in these tools may restrict using AOP in COTS-based development.

The small size of our test system, however, limits the extension of the conclusions of this study. In our future work, we plan to use a larger system with more COTS components, in order to investigate the research questions more satisfactorily.

ACKNOWLEDGMENTS

This study was associated to the INCO (INcremental COmponent based development) [7] project. Comments from our colleague Tor Stålhane gave valuable inputs to this study.

REFERENCES

1. A. Colyer and A. Clement, "Large-scale AOSD for middleware," in *Proceedings of 3rd International Conference on Aspect-Oriented Software Development*, 2004, pp. 56-65.
2. Aspectwerkz, 2004, <http://aspectwerkz.codehaus.org/index.html>.
3. C. Abts, B. W. Boehm, and E. B. Clark, "COCOTS: a COTS software integration cost model – Model overview and preliminary data findings," in *Proceedings of 11th European Software Control and Metrics Conference (ESCOM)*, 2000, pp. 325-334.
4. ComponentSource, 2004, <http://www.componentsource.com/>.
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*, LNCS 1241, Springer-Verlag, 1997, pp. 220-242.
6. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *Proceedings of 15th European Conference on Object-Oriented Programming*, LNCS 2072, Springer-Verlag, 2001, pp. 327-353.
7. INCO Project, 2000, INCO project description, Univ. Oslo/NTNU, available at: <http://www.ifi.uio.no/~isu/INCO>.
8. I. Crnkovic, B. Hnich, T. Jonsson, and Z. Kiziltan, "Specification, implementation, and deployment of components," *Communications of the ACM*, Vol. 45, 2002, pp. 35-40.
9. Java Email Server: Getting started, 2004, available at: <http://www.ericdaugherty.com/java/mailserver/gettingstarted.html>.
10. Java.util.logging, 2004, available at: <http://java.sun.com/j2se/1.4.2/docs/api/java/util/logging/package-summary.html>.
11. Jboss, 2004, available at: <http://www.jboss.org/developers/projects/jboss/aop>.
12. J. Li, F.O. Bjørnson, R. Conradi, and V. B. Kampenes, "An empirical study of COTS component selection processes in Norwegian IT companies," in *Proceedings of the International Workshop on Models and Processes for the Evaluation of COTS Components*, 2004, pp. 27-30.
13. J. Li, R. Conradi, O. P. N. Slyngstad, M. Torchiano, M. Morisio, and C. Bunse, "Preliminary result of a state-of-practice survey on risk management in off-the-shelf component-Based development," in *Proceedings of 4th International Conference on COTS-based Software System (ICCBSS '05)*, LNCS 3412, Springer-Verlag, 2005, pp. 278-288.
14. J. Voas, "COTS software – the economical choice?" *IEEE Software*, Vol. 15, 1998, pp. 16-19.
15. M. Lippert and C. V. Lopes, "A study on exception detection and handling using aspect-oriented programming," in *Proceedings of 22nd International Conference on*

- Software Engineering*, 2000, pp. 418- 427.
16. M. Torchiano and M. Morisio, "Overlooked facts on COTS-based development," *IEEE Software*, Vol. 21, 2004, pp. 88-93.
 17. P. K. Lawlis, K. E. Mark, D. A. Thomas, and T. Courtheyn, "A formal process for evaluating COTS software products," *IEEE Computer*, Vol. 34, 2001, pp. 58-63.
 18. P. J. Clemente, J. Hernández, J. M. Murillo, M. A. Pérez, and F. Sánchez, "AspectCCM: an aspect-oriented extension of the Corba component model," in *Proceedings of 28th EUROMICRO Conference*, 2002, pp. 10-16.
 19. Proposals for the architecture of COTS component intensive software systems, 2002, http://www.vtt.fi/ele/research/soh/ark/proposalsforarchitecting_ihme.pdf.
 20. R. J. Walker, E. L. A. Baniassad, and G. C. Murphy, "An initial assessment of aspect-oriented programming," in *Proceedings of 21st International Conference on Software Engineering*, 1999, pp. 120-130.
 21. S. Comella-Dorda, J. C. Dean, E. Morris, and P. Oberndorf, "A process for COTS software product evaluation," in *Proceedings of 1st International Conference on COTS Based Software Systems*, LNCS 2255, Springer-Verlag, 2002, pp. 176-187.
 22. SpamAssassin, 2004, available at: <http://eu.spamassassin.org/index.html>.
 23. SpamProbe, 2004, available at: <http://spamprobe.sourceforge.net/>.
 24. W. Schult and A. Polze, "Aspect-oriented programming with C# and .NET," in *Proceedings of 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2002, pp. 241-248.
 25. V. R. Basili and B. Boehm, "COTS-based system top 10 list," *IEEE Computer*, Vol. 34, 2001, pp. 91-93.
 26. V. R. Basili, G. Galdiera, and H. D. Rombach, "The goal question metrics approach," *Encyclopedia of Software Engineering – 2 Volume Set*, John Wiley & Sons, 1994, pp. 528-532.
 27. AspectJ Project, <http://eclipse.org/aspectj/>.
 28. Aspect-Oriented Programming with AspectJTM, 2005, available at: <http://www.cs.umd.edu/class/spring2005/cmsc838p/Slides/AOP.ppt>.

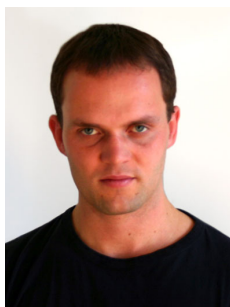
APPENDIX: AOP TERMINOLOGY

1. **Cross-cutting concern:** When different classes share the same functionality and/or requirements, we say that we have a cross-cutting concern. Usually most classes in OOP will perform a single, specific function, but they also usually have the same secondary functionality like logging and error-handling.
2. **Joinpoint:** A joinpoint is where program execution leaves one point and enters another. A program entering a function, leaving a function, or accessing a variable are examples of joinpoints. It is the point(s) in the program that you want to "point at".
3. **Pointcut:** Pointcut is used to define which jointpoints you want to target. There are two types of pointcuts. The first is the primitive pointcut, which picks out sets of jointpoints and values at those points. Another is user-defined pointcut, which is the named collection of jointpoints and values.
4. **Advice:** The advice is the code (also inside the aspect) you want to run at the jointpoints that the pointcut picks out.

5. **Aspect:** Aspect is the part of code describing how pointcuts and advice should be combined together.
6. **Weaver:** The weaver is a sort of compiler. It takes the advice and inserts the advice at the appropriate pointcuts and creates the additional code needed. It “weaves” the source-code and the aspects together.
7. **Cflow:** Cflow is one kind of pointcut. For example, $Cflow(P)$ picks out each joinpoint in the control flow of the joinpoints picked out by P .
8. **Inter-type declaration:** Aspects can declare members (fields, methods, and constructors) that are owned by other types. These are called inter-type members. Aspects can also declare that other types implement new interfaces or extend a new class.



Jingyue Li (李京悦) is presently a Ph.D. student in Computer Science at the Norwegian University of Science and Technology (NTNU). He received his master degree in Computer Science from Beijing Polytechnic University in 2001. He worked at IBM China Ltd., before he came to NTNU. His research interests include software engineering, commercial-off-the-shelf based development, open source based development, project risk management, and aspect-oriented programming.



Axel Anders Kvale received his M.S. degree in Computer Science from the Norwegian University of Science and Technology (NTNU) in 2004. He currently is working as a consultant in an IT company in Trondheim, Norway. He is working on developing the produce exchange systems and an automatic model based valuta trading clients for the financial market.



Reidar Conradi received his Ph.D. in Computer Science from the Norwegian University of Science and Technology (NTNU) in 1976. From 1972 to 1975 he worked at SINTEF as a researcher. Since 1975 he has been assistant professor at NTNU and a full professor since 1985. He has participated in many national and EU projects, chaired workshops and conferences, and edited several books. His research interests are in software engineering, object-oriented methods and software reuse, distributed systems, software evolution and configuration management, software quality, and software process improvement. He is a member of IEEE Computer Society and ACM.