A Case Study on Building COTS-Based System Using Aspect-Oriented Programming

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

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

Reidar Conradi
Department of Computer and Information Science
Norwegian University of Science and Technology
Sem Sælands vei 7-9
NO-7034 Trondheim, Norway
Simula Research Laboratory
P.O.BOX 134, NO-1325 Lysaker, Norway
+47, 73593444
conradi@idi.ntnu.no

ABSTRACT

More and more software projects are using COTS (Commercial-off-the-shelf) components. Using COTS components brings both advantages and risks. To manage some risks in using COTS components, it is necessary to increase the reusability of the glue-code so that the problematic COTS components can easily be replaced by other components. Aspect-oriented programming (AOP) claims to make it easier to reason about, develop, and maintain certain kinds of application code. To investigate whether AOP can help to build an easy-to-change COTS-based system, a case study was performed by comparing changeability between an object-oriented application and its aspect-oriented version. Results from this study show that integrating COTS component using AOP may help to increase the changeability of the COTS component-based system, if the cross-cutting concerns in the glue-code are homogenous (i.e., consistent application of the same or very similar policy in multiple places). Extracting heterogeneous or partial homogenous cross-cutting concerns in glue-code as aspects does not provide benefits. Results also show that some limitations in AOP tools may make it impossible to use AOP in COTS-based development.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software– *reuse models*

General Terms

Experimentation, Languages.

Keywords

Object-Oriented Programming (OOP), Aspect-Oriented Programming (AOP), Commercial-Off-The-Shelf (COTS) based development, component-based software development (CBSE).

1. INTRODUCTION

COTS-based development has become increasingly important in software and system development, as COTS usage promises faster time-to-market and increased productivity [13]. At the same time, COTS-based development introduces many risks. Unknown quality properties of the chosen COTS components can be harmful for the final product. Business instability of the COTS vendor may terminate the maintenance support of its COTS components [14].

To manage these risks, it is necessary to prepare for replacing current COTS components. In the process of replacing the integrated COTS components, some components relevant code (i.e., glue-code) may need to be rewritten. It is therefore important to increase the reusability of glue-code so that little effort is needed to change from one COTS component to another.

Aspect-oriented programming (AOP) is claimed to be able to increase the maintainability of a system compared to Object-oriented programming (OOP) [5]. In COTS-based development, the invocation of COTS component functionalities or methods are scattered all over the system. If cross-cutting concerns in glue-code can be separated into aspects, it will be easier to understand and change the system. To empirically investigate how to build an easy-to-change COTS-based system using AOP, a case study was performed by comparing the changeability in an object-oriented system and its aspect-oriented version.

Results from our study show that proper use of aspect-oriented programming in COTS component integration can help to increase the changeability of the system. Result also show that detailed plan and design should be performed before the decision of using AOP in COTS-based development.

The rest of this paper is organized as follows: Section 2 introduces some related concepts and background of this study. Section 3 describes our research design. Section 4 presents the results and Section 5 discusses them. Conclusion and future work are presented in section 6.

2. BACKGROUND

2.1 COTS Component Definition

The essential question for COTS component-based development is "What do you mean by a COTS component?" We have used the definition by Torchiano and Morisio [16], where a COTS component:

- 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, 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 available.

The granularity of the COTS software can be different. In this study, we focus on COTS "components". 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 is, a component is an "executable unit of independent production, acquisition, and deployment that can be composed into a functioning system." This definition means that we include not only components following COM, CORBA, and EJB standards, but also C++ or Java^{TM} libraries. This definition is consistent with the scope in the component marketplace [4].

2.2 Risks in COTS Component-Based Development

A COTS component-based process consists of four phases, comprising [3]:

- COTS component assessment and selection
- COTS component tailoring
- COTS component integration
- Maintenance of COTS and non-COTS parts of the system

COTS-usage promises advantages, but also brings many possible risks [13, 14]. Proper risk management is needed in each phase:

- In the COTS component selection and evaluation phase

One risk in this phase is that wrong components may be selected. Several formal selection processes and decision making methods have been proposed to support the selection and evaluation of COTS components [21]. In some of these proposed selection processes, hands-on trial is regarded as a necessary step [12, 17]. Hands-on trial means to build glue-code and integrate 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 cross-cutting concerns in glue-code spread throughout the system. To change the testing from one COTS component to another, a lot of glue-code needs to be modified and rewritten

- In the COTS component tailoring and integration phase

One risk in this phase is that too much effort needs to be spent on solving the mismatch between COTS components. As COTS components may be bought from different vendors, the internal implement may cause the mismatch between those components. A lot of glue-code may be needed to integrate these COTS components and make them work together.

- In the maintenance phase

One risk in this phase is that vendor may go bankrupt and fail to give support to the current COTS component running in the system. Some vendors may withdraw support on the old version component when they publish the new version. The new version component may have no backward compatibility with the old version one.

One solution to the above risks is to try to build an easy-to-change COTS component-based system. It means that the COTS component users are not bound to specific COTS components and a specific COTS vendor. If the selected COTS components bring unexpected problems, they can easily be substituted by other components.

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 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. 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. As a result, a single aspect can contribute to the implementation of a number of procedures, modules, or objects. It is therefore help to increase reusability of the source code [5]. Several different AOP tools have been built, such as AspectJ [6], AspectWerkz [2], and JBoss AOP [11].

3. RESEARCH DESIGN

3.1 Motivation and Research Questions

Most current glue-code is built using OOP. The advantage of using AOP over OOP is that it is possible to modularize glue-code that cross-cut the whole application. In COTS-based development, the invocation of COTS component functionalities or methods are scattered throughout the system. If cross-cutting concerns in glue-code 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]. There were, however, few studies on integrating COTS component (where the source code is either not available or hard to modify) using AOP [19]. The motivation of this study is to empirically investigate whether AOP can help to build an easier-to-built and easier-to-change COTS component-based system than OOP.

Thus, our first research question **RQ1** is to compare how much effort is needed to integrate a COTS component by AOP vs. OOP.

- ***RQ1: Is it easier to build a COTS-based system using AOP than using OOP?***

The second research question **RQ2** is to compare how much effort is needed to change from one COTS component to another.

- ***RQ2: Is it easier to change a COTS-based system built by AOP than a system built by OOP?***

3.2 System and Programming Language Selection

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 selected to re-engineer an existing system because:

- It will be easier to compare the results since the systems are identical except that some parts were extracted into aspects in the AOP version
- The disadvantages of building a system from scratch by both OOP and AOP is that the measurements might be influenced by choices made by the developer, and not by the difference in AOP vs. OOP (i.e. a specific problem is solved elegantly in the OOP model and poorly in the AOP model). This might occur on both the modelling level and the implementation level.

The system chosen for the study is an open source Java Email Server, the JES server [9]. It is built by OOP principles. Some objects in this application are encapsulated as components, such as logging, spam-checking, etc.

Although the source code of components in JES server is available, we treated these components as COTS components in this study, i.e. we did not change source code inside components. Code relevant to these components is regarded as glue-code.

Because the aspect code is combined with the primary programming code by an aspect weaver, it is important that the AOP tool can do byte code weaving because most COTS components are delivered in byte code format. The aspect-oriented tool we selected is AspectJ version 1.1 [6]. AspectJ extends Java^{TM} and supports byte code weaving. It is therefore possible for us to weave an existing byte code COTS component without source code.

3.3 Research Steps

There are four steps in this study.

3.3.1 Re-engineering the glue-code of the logging component using AOP

The **first** step was to re-engineer the JES server using AspectJ. We first selected an existing component in the JES server and re-engineered the glue-code using AOP. In the JES server, almost all classes utilize the *log4j* component for logging, and make it a cross-cut concern. By moving the glue-code relevant to the *log4j* into a separate aspect, all classes become independent of this component.

3.3.2 Add glue-code to integrate an additional spam-checking COTS component

The **second** step was to investigate research question **RQ1.** To investigate the efficiency of adding a component, a spam-checking COTS component (i.e., *SpamAssassin* [22]) was added to both the OOP and the AOP system. *SpamAssassin* is a popular spam-checker for most email servers. The JES server uses the class SMTPMessage to keep the email and functions related to an email message.

- In the OOP version, the spam-checking routine is created inside the SMTPMessage class. When the SMTPProcessor class has received a new e-mail and stored it inside a SMTPMessage, the spam-checking is called in the SMTPMessage, and the original message with altered headers will be returned as the result.
- In the AOP version, the *SpamAssassin* extension is implemented inside an aspect. The routine for checking a SMTPMessage is the same as in the OOP version, but the spam-checking routine is called inside the aspect. A pointcut picks out all the places where a SMTPMessage should be checked for spam. The aspect calls *SpamAssassin* and alters the SMTPMessage. The result is that every SMTPMessage that created by SMTPProcessor is checked with *SpamAssassin* without the knowledge of SMTPProcessor and SMTPMessage. It is all done inside an aspect.

3.3.3 Replace logging component with another logging COTS component

The **third** step was to investigate research question **RQ2**. We used another logging component to replace the current logging component in both the OOP and AOP version of the system. In the Java^{TM} Development Kit (SDK) version 1.4, there is a new logging-system available (i.e., *util.logging* [10]). It is for logging and is built up in the same way as *log4j*. We therefore decided to replace the *log4j* component with the *util.logging* package from JDK 1.4. There are three steps in both the AOP version and the OOP version:

- The first change to do 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*, this is done through an xml-file that it passed to the system by the Java^{TM} command that starts the system.

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

3.3.4 Replace SpamAssassin component with another COTS component

The **fourth** step was to investigate research question **RQ2** further. We used another spam-checking component (*SpamProbe* [23]) to replace the *SpamAssassin* component in both the OOP version and AOP version of the system. *SpamProbe* is a spam detection application using Bayesian analysis of terms contained in the email. It works in a way similar to *SpamAssassin*. The email server needs to call *SpamProbe* and ask it to scan the email. The difference between *SpamAssassin* and *SpamProbe* is their output. The output received from *SpamProbe* is the result of the conducted scan (the additional headers). The output received from *SpamAssassin* is the original message with altered headers. The changes in the OOP version and AOP version of the system are as follows:

- In the OOP version, each class that scans the message has to be changed. The result from the *SpamProbe* needs to be read and appended to the message-header.
- In the AOP version, the change is the same as in the OOP version. However, only the aspect is changed, none of the classes in the system needs to be changed, regardless of where the spam-checking is required.

4. RESULTS AND LESSONS LEARNED

To compare the changeability between the AOP version and OOP version of the system, our metrics records how many LOC and classes needed to be modified.

4.1 Results of Research Question RQ1

To add the spam-checking component *SpamAssissin*, the total lines-of-code (LOC) and number of classes were changed (added, modified or deleted) in the OOP version and AOP version are showed in the following Table 1.

Table 1. Changes performed to add SpamAssassin

Changes	**OOP Version**	**AOP version**
LOC changed	36	44 (In aspect)
Number of classes changed	2	0 (1 aspect changed)

Since there are only two classes were changed in order to add the *SpamAssissin* in the OOP version, the AOP version needed to add more LOCs. It is because AOP version needed extra LOCs to define pointcuts. If several classes need to be changed to add *SpamAssissin*, AOP version would have a benefit since only a new pointcut definition is needed for each additional class. In the OOP version, the same functionality needs to be implemented in each class.

4.2 Results of Research Question RQ2

In the process of replacing the *log4j* component with *util.logging*, the total lines-of-code (LOC) and number of classes were changed (added, modified or deleted) in the OOP version and AOP version are showed in the following Table 2.

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

Changes	**OOP Version**	**AOP version**
LOC changed	184	162 (In aspect)
Number of classes changed	12	0 (1 aspect changed)

We can see that the LOC changed in OOP system and AOP system are almost the same. The reason is that ***glue-code spreading in the system is not homogenous (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 version as showed in Figure 1.

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

Figure 1. Code for logging in the OOP version.

By printing out these static strings, the system gives a reasonable clue of what it did (or failed to do).

With the general logging in the AOP example, this cannot be accomplished. The logging will be limited to the information provided by the joinpoint (name of the function, name of the enclosing function, arguments, class name etc). It is possible to be as accurate with logging in AOP as with OOP, but this require us to define each and every pointcut where we want to log and treat these joinpoints individually as Figure 2.

The result was that we had to build several quite complex pointcuts to define where we want to log. If the cross-cutting concern is homogenous, there would be a benefit in the AOP version when measuring LOC changed.

The value of the AOP version in this case is that only the logger-aspect was changed. In the OOP version all the classes (12 classes) using the *log4j* system had to be changed.

```
//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);  }
```

Figure 2. Code for logging in the AOP version.

In the process of changing the *SpamAssisin* component with *SpamProbe*, the total lines-of-code (LOC) and classes that need to be changed (added, modified or deleted) in the OOP version and AOP version are showed in the following Table 3.

Table 3. Changes performed to replace SpamAssisin with SpamProbe

Changes	**OOP Version**	**AOP version**
LOC changed	15	15
Number of classes changed	1	0 (1 aspect changed)

In the OOP version, SMTPMessage was changed. The routine that scan the message was changed to use *SpamProbe* instead of *SpamAssassin*. In the AOP version, only advices were changed when changing from SpamAssisin to *SpamProbe*, regardless of where the spam-checking functionality is called. The system doesn't even need to know which spam-checking component is used. In the OOP version, all classes calling the spam-checking functionality were modified.

4.3 Lessons Learned in Re-engineering

When we implemented the aspect-oriented system using AspectJ, some unexpected limitations of AspectJ version 1.1 made it difficult to use AOP in the COTS-based development. The reason is that all COTS components are supposed to be not changeable. 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 method
- 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 in Figure 3, we need to combine several pointcuts as showed in Figure 4.

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

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

```
//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);
```

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

AspectJ version 1.1 does not support to get a reference of the variable if there is no joinpoint in the cflow that has accessed the according variables before.

For example, if we use AOP to build the glue-code as showed in Figure 5, it is not possible to get a reference to *s*, *address* and *user* together, because no joinpoint (or **cflow**) used all these three variables at the same time. We therefore cannot access both *s*, *address*, and *user* together.

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

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

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

5. DISCUSSION

5.1 Comparison AOP with OOP in COTS-Based Development

Comparing with OOP, our results show that there are both benefits and limitations of using AOP in COTS-based system.

The main benefit is that fewer classes need to be changed when adding and replacing COTS components (see Table 1, 2 and 3). It is because most changes needed are centralized in aspects instead of being scattered throughout the system. However, using AOP does not ensure that fewer LOC need to be modified than using OOP when adding or replacing COTS components.

- If the possible glue-code includes homogenous cross-cutting concern in several classes, the LOC to be changed when adding or replacing the component may be fewer in the AOP system. The reason is that AOP removes the dependencies between the classes and the COTS component. It makes the system oblivious about the existence of the COTS component.
- If the cross-cutting concerns in the glue-code are (partly) heterogeneous as showed in section 4.1 and section 4.2, more LOC may be needed in the AOP version if we want to add COTS components in the system. It is because every occurrence of the concern must be defined using a joinpoint and advice. In case of changing the COTS component, almost equal amount of LOC need to be changed in the AOP and OOP version.

Most COTS-based systems were built using OOP principles. Languages and tools support to build a COTS-based system are advanced and completed. However, AOP tools are still immature and limited. Because most COTS components are delivered as byte code instead of source code, AOP tool should be able to weave the byte code. However, current tools support byte code weaving, such as AspectJ and AspectWerkz, are based only on Java^{TM}.

Other limitations of AOP tools (see section 4.3) prohibit using AOP in COTS-based development, because they require modifications inside the COTS component.

5.2 Comparison with Related Works

Some previous studies have empirically investigated how to use AOP in different kind of applications:

- Walker et al. have empirically 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 in two systems (one is built with AOP and another is built by OOP) with 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. In our study, we re-engineered an OOP system into an AOP system and compared them. This avoids the possible bias caused by differences in system design (i.e., good design in OOP and bad design in AOP). Our results give further support to their conclusion. If the COTS-based system we developed using AOP, it is easier to reason about and change if the interface between a COTS component and other part of the system is completely separated. Other parts of the system are even oblivious about the existence of the COTS component.
- Lippert et al. investigated the benefits of AOP by re-engineering an OOP system and extracting exception detection and handling as aspects [15]. They concluded that AOP provides better support for reuse. While they worked with a system that can be changed completely, our study focuses on a system where the code in the COTS components cannot be changed.
- 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 crosscutting concerns in the middleware. Our results in changing a logging component give further support to their conclusion (i.e., it is more challenging to re-engineer heterogeneous cross-cutting concerns than homogeneous ones). They proposed processes and methods that can help to change the heterogeneous crosscutting concerns into ideal aspects. In their system, all source code can be changed. In the COTS-based system, we may not be able to extract some heterogeneous crosscutting concerns into good aspects, because the COTS component is not changeable.
- Other studies tried to integrate the AOP into a component model, such as CORBA, and .NET [18, 24]. Here, the COTS components must follow these new component models. However, there are still few COTS components in the market that follow these new component models. Our study is therefore limited to the COTS component in the form of Java^{TM} libraries.

5.3 Possible Threats to Validity

The threat to **internal validity** of this study is that the OOP version has been re-factored several times to improve the design and implementation while the AOP version has not. It is most likely that the OOP version has a very good design compared to the AOP version.

The threat to **construct validity** is that we used LOC and number of classes changed (added, deleted, or modified) to measure the changeability of the system. There are several metrics proposed to measure the changeability of the OOP system. 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 on AOP and OOP. We therefore selected more objective metrics (i.e., LOC and number of classes changed).

The possible threat to **external validity** of this study is that the size of our system is not very huge. However, the results of this study discovered some important issues in using AOP in COTS-based development.

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

6. CONCLUSION AND FUTURE WORK

We have studied how AOP ease the adding and replacement of components in COTS-based development. We re-engineered an existing OOP application using AOP and compared the LOC and number of classes needed to be changed in order to add and replace COTS components. From this study, we found that:

- When adding or replacing a COTS component, the main benefit of using AOP in COTS-based is that fewer classes need to be changed than using OOP. However, using AOP does not ensure that fewer LOCs need to be modified than using AOP when add or replace COTS components. It depends on whether glue-code is homogenous or not. Using AOP when glue-code is (partly) heterogeneous may not bring benefits. A careful analysis on cross-cutting concerns in the glue-code is therefore needed before the decision of using a certain COTS component.
- To integrate COTS components using AOP, the aspect 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 conclusions of this study. In our future work, we plan to use a larger system with more COTS components, in order to investigate research questions more satisfactorily.

7. ACKNOWLEDGEMENTS

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.

8. REFERENCES

[1] Adrian Colyer, Andrew Clement, 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.

[2] Aspectwerkz: http:// aspectwerkz.codehaus.org/index.html

[3] Chris Abts, Barry W. Boehm, and Elizabeth Bailey Clark: COCOTS, A COTS Software Integration Cost Model - Model Overview and Preliminary Data Findings. In *Proceedings of the 11th European Software Control and Metrics Conference* (ESCOM 2000) (Munich, Germany, April, 2000), 325-33.

[4] ComponentSource: 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* (Jyväskylä, Finnland, June, 1997), Springer Lecture Notes in Computer Science, Vol. 1241, 220-242.

[6] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mil Kersten, Jeffrey Palm, and Willian G. Griswold, An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming* (Budapest, Hungary, June 18-22), Springer Lecture Notes in Computer Science, Vol. 2072, 327 – 353.

[7] INCO project: http://www.ifi.uio.no/~isu/INCO/

[8] Ivica Crnkovic, Brahim Hnich, Torsten Jonsson, and Zeynep Kiziltan, Specification, Implementation, and Deployment of Components. *Communication of the ACM*, 45, 10 (October, 2002), 35-40.

[9] Java Email Server: Getting started: http://www.ericdaugherty.com/java/mailserver/gettingstarted.html

[10] Java.util.logging: http://java.sun.com/j2se/1.4.2/docs/api/java/util/logging/package-summary.html

[11] Jboss: http://www.jboss.org/developers/projects/jboss/aop

[12] Jingyue Li, Finn Olav Bjørnson, Reidar Conradi, and Vigdis By 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*, (Edinburgh, Scotland, May, 2004), IEE ISBN 0-86341-422-2, 27-30.

[13] J. Voas, COTS Software – the Economical Choice?. *IEEE Software*, 15, 2 (March/April, 1998), 16-19.

[14] J. Voas, The Challenges of Using COTS Software in Component-Based Development. *IEEE Computer*, 31, 6 (June, 1998), 44-45.

[15] Martin Lippert and Cristina Videira Lopes, A Study on Exception Detection and Handling Using Aspect-Oriented Programming. In *Proceedings of the 22nd International Conference on Software engineering* (Limerick, Ireland, June, 2000), IEEE Computer Society Press, 2000, 418 – 427.

[16] M. Torchiano and M. Morisio, Overlooked Facts on COTS-based Development. *IEEE Software*, 21, 2 (March/April, 2004), 88-93.

[17] Patricia K. Lawlis, Kathryn E. Mark, Deborah A. Thomas, and Terry Courtheyn, A Formal Process for Evaluating COTS Software Products. *IEEE Computer*, 34, 5 (May, 2001), 58-63.

[18] Pedro J. Clemente, Juan Hernández, Juan M. Murillo, Miguel A. Pérez, Fernando Sánchez, AspectCCM: An Aspect-Oriented Extension of the Corba Component Model. In *Proceedings of the 28th Euromicro Conference*, (Dortmund, Germany, September 2002), IEEE Computer Society Press, 2002, 10-16.

[19] Proposals for the architecture of COTS component intensive software systems: http://www.vtt.fi/ele/research/soh/ark/proposalsforarchitecting_ihme.pdf

[20] Robert J. Walker, Elisa L.A. Baniassad, and Gail C. Murphy, An Initial Assessment of Aspect-Oriented Programming. In *Proceedings of the 21st International*

Conference on Software Engineering (Los Angeles, California, United States, May, 1999), IEEE Computer Society Press, 1999, 120 – 130.

[21] Santiago Comella-Dorda, John C. Dean, Edwin Morris, and Patricia Oberndorf, A Process for COTS Software Product Evaluation. In *Proceedings of the First International Conference on COTS Based Software Systems (*Orlando, FL, USA, February 4-6, 2002*)*, Springer Lecture Notes in Computer Science, Vol. 2255, 176-187.

[22] SpamAssassin: http://eu.spamassassin.org/index.html

[23] SpamProbe: http://spamprobe.sourceforge.net/

[24] Wolfgang Schult and Andreas Polze, Aspect-Oriented Programming with C# and .NET. *In Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* (Washington D.C, United States, April, 2002), IEEE Computer Society Press, 2002, 241-248.