Comparative Evaluation of Programming Paradigms: Separation of Concerns with Object-, Aspect-, and Context-Oriented Programming

Fumiya Kato, Kazunori Sakamoto, Hironori Washizaki, and Yoshiaki Fukazawa

Dept. Computer Science and Engineering
Waseda University
Tokyo, Japan
E-mail: fum_kato@asagi.waseda.jp

Abstract

There are many programming paradigms for the separation of concerns (SoC). Each paradigm modularizes concerns in a different way. Context-oriented programming (COP) has been developed as a supplement to object-oriented programming (OOP), which is one of the most widely used paradigms for SoC. It modularizes concerns that are difficult for OOP. In this paper, we focus on three paradigms - OOP, aspect-oriented programming (proposed as a supplement to OOP that has a different approach from COP), and COP - and study whether COP can modularize concerns better than other two paradigms in given situations. Then we determine the reasons why COP can or cannot better modularize concerns.

1. INTRODUCTION

In software development, the separation of concerns (SoC) is an important matter. To deal with SoC, numerous programming paradigms have been proposed. Object-oriented programming (OOP) is one of the most widely used paradigms. However, some concerns called cross-cutting concerns (CCCs) are difficult to modularize for OOP and are often scattered over modules in a program. As a supplement to OOP, aspect-oriented programming (AOP) has been proposed. AOP modularizes CCCs as *aspects* that weave codes into other modules.

In recent years, context-oriented programming (COP) has been proposed as a supplement to OOP that has a different approach from AOP. COP modularizes behavior that depends on the state of execution as *layers*. Several COP languages have been developed [1, 2, 3] however, there has been little research on the situations COP is most effective for developing and how COP modularizes concerns better than the other paradigms.

In this paper, we focus on three programming paradigms: OOP, AOP, and COP and perform comparative experimentation on these paradigms to research their effectiveness of achieving SoC. To measure the effectiveness of achieving SoC, we perform experimentation in terms of the description amount and the locality of change. The purpose of this study is to answer the following research questions (RQs):

RQ1: Do the description amount and the locality of change differ in implementing programs with the same requirement by OOP, AOP, and COP?
RQ2: Are there any situations in which COP is superior or inferior to OOP and AOP in terms of the description amount and the locality of change?
RQ3: What features of COP result in its superiority to OOP and AOP?
RQ4: What features of COP result in its inferiority to OOP and AOP?

The contributions of this paper are as follows:

- **Suggestion of the comparative evaluation scheme for programming paradigms.**
- **The comparative results of the superiority or inferiority of OOP, AOP, and COP in terms of the description amount and the locality of change.**
- **Specification of the causes of superiority and inferiority in the comparative results.**

The rest of the paper is structured as follows. Section 2 introduces the cross-cutting concerns that are difficult to modularize by OOP, and how AOP and OOP modularize such concerns. Section 3 presents the study format: the purpose of study, the method of comparative study, and the programming languages used. Section 4 shows the results of the study and our analysis. We discuss related works in Section 5 and summarize the paper in Section 6.

2. BACKGROUND

2.1. Cross-Cutting Concerns

CCC is a type of concern that is entangled with other concerns. As the source code level, CCCs are often scattered over modules in a program. For instance, logging to

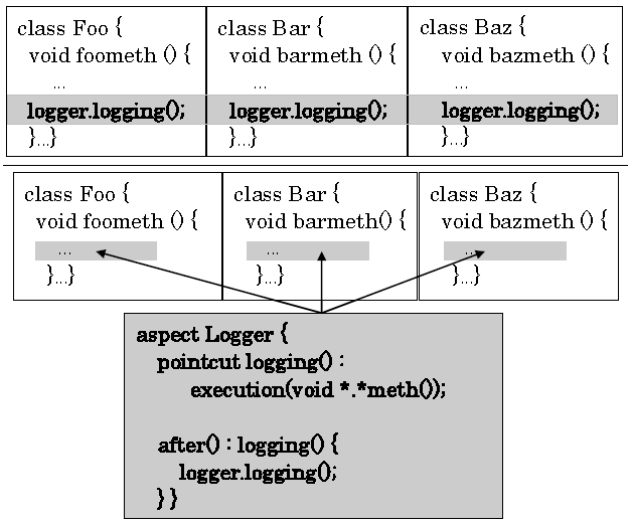

Figure 1. Cross-cutting concern of logging code in OOP (top) and modularization of CCC of logging code in AOP (bottom)

carry out debugging is one CCC. As shown in the top of Figure 1, it is difficult to modularize CCCs by OOP. A logging code can be written in many modules in a program in OOP. Such a situation worsens maintainability, because scattered logging codes make programs unreadable, and causing mistakes in modifying or deleting logging codes.

2.2. Aspect-Oriented Programming

AOP has been proposed as a supplement to OOP. AOP modularizes CCCs as *aspects*. The bottom of Figure 1 shows the modularization of CCCs about logging by AOP. Aspects separate CCCs from the major concerns that each class modularizes by weaving CCC codes into other modules. AOP has a *pointcut-advice* mechanism for achieving weaving. *Advice* defines the operation woven into other modules. *Pointcut* defines the modules into which advice is woven and the points - for instance, particular methods are executed or particular types of objects are accessed - at which woven codes are validated.

2.3. Context-Oriented Programming

COP has been proposed as a supplement to OOP via a different approach from AOP. COP can modularize CCCs about changing behavior depending on the state of execution as *layers*. A layer defines the methods in other classes. Methods defined in a layer are executed in a particular state instead of the original method definitions. Figure 2 shows the modularization of CCCs about logging by COP. A layer defines the methods in other classes with logging codes. Method definitions with logging codes in a layer can be executed instead of the base definitions in other classes when

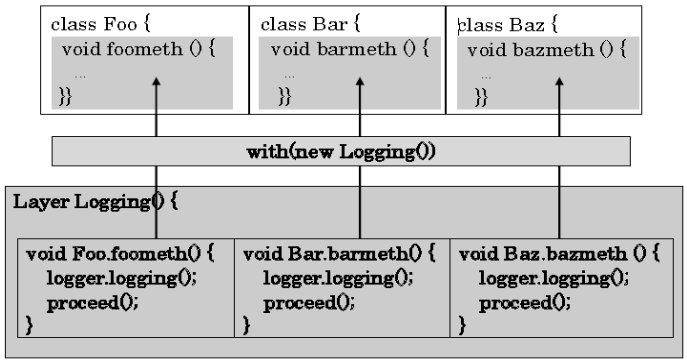

Figure 2. Modularization of CCC with layer

the layer is activated. Layers are activated by *with* statements. In addition, base definitions can be called from the definitions in a layer by *proceed* statements.

3. COMPARATIVE EVALUATION SCHEME

Figure 3 shows the overview of the comparative evaluation. To detect situations in which COP is superior or inferior to OOP and AOP, we performed comparative experimentation. In this paper, we created three sample programs containing CCCs about changing behavior, implemented them in three programming languages, and performed seven modification experimentation. In addition, we implemented some parts of open-source software (OSS) in Java with each programming language. From the results of our implementation, we discuss the superiority and inferiority of COP.

In the analysis of the results, we use two criteria: description amount and locality of change. For comparison of the locality of change, we created seven change tasks as follows: add classes related or unrelated to CCCs, add methods related or unrelated to CCCs, rename methods related or unrelated CCCs, and delete CCCs. We counted the chunks of code that the change tasks forced to modify.

As programming languages for OOP, AOP, and COP, we use Java, AspectJ [4, 5] , and JCop [1] respectively. AspectJ and JCop are implemented by extending Java. Thus, the forms of the fundamental descriptions of Java, AspectJ, and JCop are similar. Therefore, the results should be less affected by differences in the abstraction level and grammar of languages, and more affected by differences among paradigm features.

4. EVALUATION EXPERIMENTATION

4.1. Target

4.1.1. Sample Programs

To compare OOP, AOP, and COP, we created three sample programs that are based on the samples on the JCop project page [6]: address book, bank account system, and

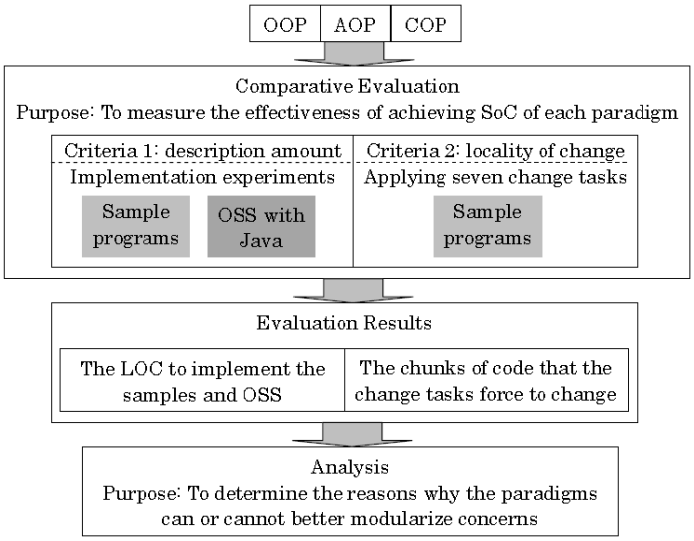

Figure 3. Overview of the comparative evaluation.

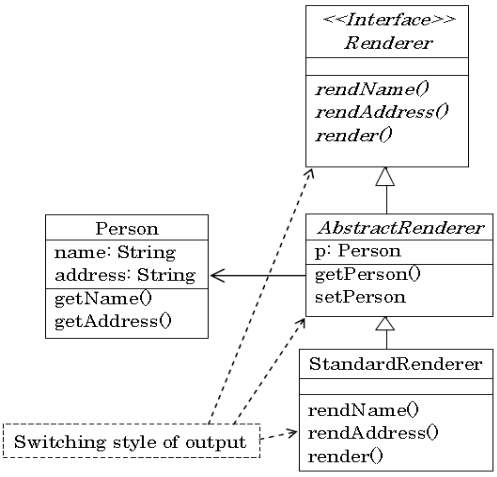

Figure 4. Design of address book and additional design concern

```
Fumiya KATO
Fumiya KATO, Chiba Japan
<div class = "person"><b>*Fumiya KATO*</b></div>
<div class = "person"><b>Fumiya KATO</b></div>, <i>Chiba Japan</i>
```

Figure 5. Output of address book: each option is switched independently

BMI calculator (https://github.com/FumKato/CompParadigm). They contain CCCs about changing behavior of several classes depending on the state of execution.

Figure 4 shows the design of the address book. As the base behavior, the StandardRenderer class implementing a Renderer interface outputs the fields of a Person object. The CCCs about changing behavior enable renderer classes to switch the styles of output: to render a name with an address, to add HTML tags, and to decorate outputs by ' * ' and ' _ '. As shown in Figure 5, each output option is switched independently. As the description amount, the lines of code (LOC) to implement the program are counted. As the change of locality, the seven change tasks defined in Section 3 are applied to the CCCs about output options. For instance, *Add-Related-Class* adds a new renderer class so that output options can be set, and *Add-Unrelated-Class* adds a new class unrelated to output options.

In the same way as the address book, bank account system has CCCs: logging and encryption of data. The BMI calculator, which is a GUI application with a Qt Jambi framework [8], also has CCCs: switching units of input and style of output.

4.1.2 Open-Source Software

We perform a comparative study of sample programs prior to the comparative study of OSS. The results and analysis of the samples, presented in a later section, indicate that a correspondence relation between layers in JCop and the decorator pattern - one of the GoF design patterns - in Java exists. If JCop improves implementation in the decorator pattern, we can propose the part of the programs the decorator pattern is applied as the COP usage guideline. The guideline can be used to recommend to developers the more positive use of COP than of OOP. Then we performed a comparative study of OSS in the Java that decorator pattern are applied. The OSS we used is JHotDraw 5.3 [7], a Java GUI framework for graphics editor. We found four places decorator patterns are applied, and rewrote them in AspectJ and JCop.

4.2. Evaluation Results

The top of Figure 6 shows the LOC to implement sample programs in each programming language. In the three sample programs, implementation in Java needs the most LOC, and implementation in JCop needs the least LOC.

Table 1 shows the change tasks and chunks of code that the tasks forced to change. In *Add-Related-Method*, *Add-Unrelated-Method*, and *Rename-Unrelated-Method*, implementation in AspectJ and JCop need less change than in the case of OOP. In particular, in *Add-Related-Method* of the address book and bank account system, implementation in JCop needs less change than in the other languages. On the other hand, in *Add-Related-Method*, JCop has the largest number of changed chunks of code.

The modularization of CCCs about changing behavior in sample programs is achieved using the decorator pattern in Java, aspects in AspectJ, and layers in JCop.

The bottom of Figure 6 shows the LOC of implementation where the decorator pattern is applied in each programming language. In three of the four cases, implementation in JCop needs the most LOC, and in the other case, it is impossible for implementation in JCop to rewrite the codes

Table 1. Seven change tasks and affected chunks of code. ' n ' indicates the number of times the tasks, e.g. in *Add-Related (Unrelated)-Class*, ' n ' indicates the number of added classes (n > 0).

	Address Book			Bank Account System			BMI Calculator		
	OOP	AOP	COP	OOP	AOP	COP	OOP	AOP	COP
Add-Related-Class	n	0	3n	2n	0	6n	13n	4n	2n
Add-Unrelated-Class	0	0	0	0	0	0	0	0	0
Add-Related-Method	3n	4n	2n	2n	2n	2n	3n	3n	2n
Add-Unrelated-Method	n	0	0	2n	0	0	2n	0	0
Rename-Related-Method	3n	2n	2n	2n	2n	2n	2n	2n	2n
Rename-Unrelated-Method	3n	0	0	2n	0	0	2n	0	0
Delete-Concerns	n	0	1	n	0	1	6n	0	0

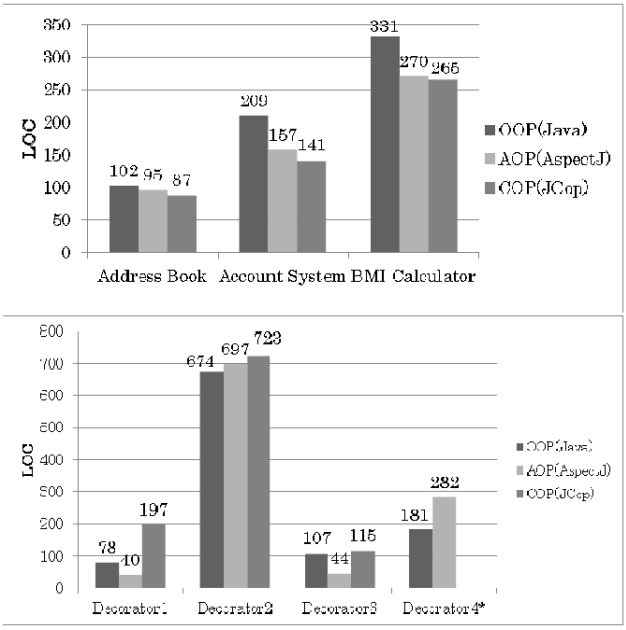

Figure 6. LOC for sample program (top) and OSS with the decorator pattern (bottom) implementation in three programming languages

applied the decorator pattern without changing the fundamental structure of the program. Implementation in AspectJ can reduce LOC in two of the four cases, and in the other cases LOC increases compared with that in OOP.

4.3. Discussion

RQ1: Do the description amount and the locality of change differ in implementing programs with the same requirement by OOP, AOP, and COP?

The comparative result shown in Figure 6 indicates that the description amount differ in implementing programs with the same requirement by OOP, AOP, and COP. The results shown in Table 1 indicate that the localities of change also differ. Therefore, using the programming paradigms properly in appropriate situations would improve the software quality from the viewpoint of the description amount and the locality of change compared with that in the case of using a single paradigm.

RQ2: Are there any situations in which COP has superiority or inferiority to OOP and AOP in terms of the description amount and the locality of change?

The comparative results of the sample programs shown in the top of Figure 6 indicate that situations in which COP is superior to OOP and AOP exist from the viewpoint of the description amount. On the other hand, the results for OSS shown in the bottom of Figure 6 indicate that situations in which COP is inferior to OOP and AOP exist.

The results shown in Table 1 indicate that situations in which COP is both superior and inferior to OOP and AOP exist from the viewpoint of the locality of change. The situations in which COP is superior to OOP are the change tasks of *Add-Unrelated-Method*, *Rename-Unrelated-Method*, and *Delete-Concerns*. These change tasks affect COP in the same way as AOP. In *Add-Related-Class*, COP is less affected than OOP and AOP.

RQ3: What features of COP result in its superiority to OOP and AOP?

The situations in which COP is superior to other two paradigms can be classified under two types of descriptions: necessary for Java but unnecessary for AspectJ and JCop, and necessary for AspectJ but unnecessary for JCop.

First, we discuss the former. As mentioned above, the decorator pattern is applied in implementation in Java. Figure 7 shows that the concerns about changing behavior are implemented as the Html class; such a class is called *decorator class*. The Html class changes the behavior of StandardRenderer class - such classes are called *component classes* - through the field of a component class object. Therefore, decorator classes need descriptions that set and get component class objects. On the other hand, implementation in AspectJ or JCop does not need such descriptions, because aspects and layers have mechanisms of weaving codes that change behavior into other classes.

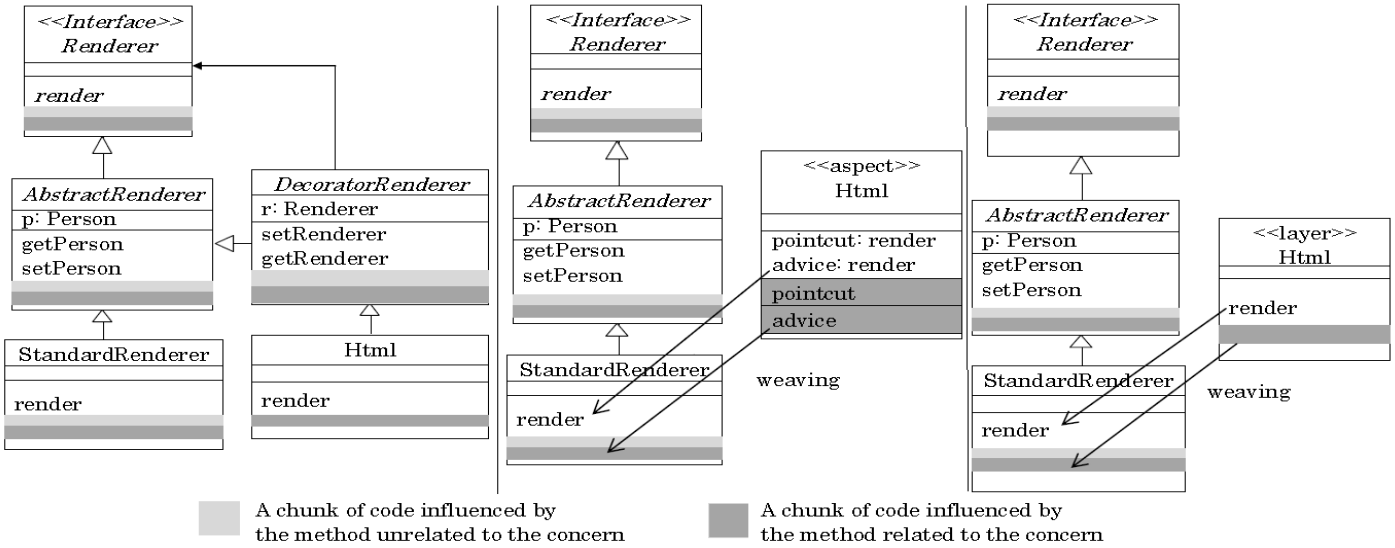

Figure 7. Part of address book program implemented in Java (left), AspectJ (center), and JCop (right).

Furthermore, a decorator class needs to implement the same interface with a component class in implementation in Java. Therefore, a decorator class needs to define not only the methods that change behavior but also other methods unrelated to the concerns about changing behavior. On the other hand, aspects and layers do not need to define the methods unrelated to the concerns about changing behavior, because if aspects or layers do not describe the methods of other classes, the methods defined in each class are just called. For that reason, adding methods unrelated to the concerns about changing behavior does not force aspects or layers to change any codes and only implementation in Java need to change codes as shown in Figure 7.

Secondly, we present the later. Figure 7 shows the example. Adding a method that changes behavior forces decorator classes, aspects, and layers to change codes. To adopt the added method, aspects need to define advice - that is the operation weaved into other classes - and the pointcut that defines the classes and timing advice is weaved for each advice. Therefore, two chunks of code - pointcut and advice - are needed for each new method that changes behavior. On the other hand, the method definitions in layers are bound to only classes weaved codes. The timing at which weaved codes are validated is bound to layers. Therefore, if the timing of validating weaved codes is common to the methods defined in a layer, JCop defines it only once no matter how many methods are defined in a layer. For these reasons, implementation in JCop needs less change in the *Add-Related-Method* than that in AspectJ.

RQ4: What features of COP result in its inferiority to OOP and AOP?

The situations in which COP shows inferiority originate from the change task *Add-Related-Class* in Table 1 and the results of the decorator pattern in OSS shown in Figure 6. Figure 8 shows an example: part of the address book program. In the change task *Add-Related-Class*, the TableRenderer class is added. The TableRenderer class has methods that changes behavior the same as the StandardRenderer class. This change task does not force an aspect to change any codes, because the pointcut rendering in the Html aspect already defines the classes that are weaved codes as subclasses of the Renderer interface. Such a definition is achieved by ‘ Renderer+ ’ .

On the other hand, *Add-Related-Class* forces layers to describe redundant method definitions as shown in Figure 8. An Html layer needs to describe almost the same method definitions that differ only in class path to weave into the StandardRenderer and TableRenderer classes, because each method definition is bound on only one class. Therefore, situations in which COP is inferior to OOP and AOP exist.

The main reasons why implementation in JCop needs more LOC than that in other languages in OSS are the same as those for sample programs. A decorator class is often applied to two or more component classes; therefore, implementation in JCop needs redundant method definitions such as in the example given in Figure 8.

In addition, layers in JCop cannot define members that are not weaved into other classes. Therefore, in Java, if decorator classes define private members, layers cannot define such members. Thus, each class that is weaved code by layers needs to define members that are accessed by only layers. For this reason, implementation in JCop makes codes accessed by only layers scattered.

4.4. Threats to Validity

The results and analysis are based on our implementation in each programming paradigm. Therefore, threats to internal validity exist, because the affect of the difference in the developer on the comparative results is not discussed in this paper. As future work, we will perform implementation experimentation with several programmers to validate

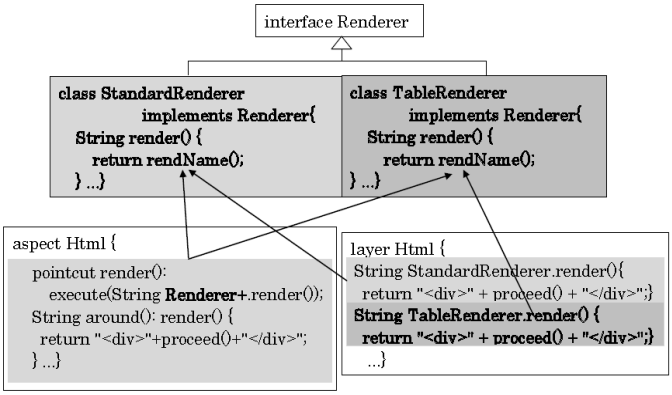

Figure 8. Situation in which COP has inferiority to OOP and AOP. Adding a new related class forces layers to describe redundant method definitions that differ only in class path.

the generality of the results and analysis discussed in this paper.

5. RELATED WORK

In recent years, many COP languages have been developed [9] including those for Java [1, 2, 3, 10]. These studies focused on specification of the languages and their performance of execution. However, these studies do not evaluate how degree and what situations these languages can achieve SoC compared with other programming paradigms.

The prior works that have guided this study is given in [11, 12, 14, 15]. Figueiredo et. al [11] performed a quantitative study of AOP that investigated the efficacy of AOP to prolong design stability of software product lines. This study focused upon a multi-perspective analysis in terms of modularity, change propagation, and feature dependency measured by metrics for concerns [13]. Kiczales and Mezini [14] performed comparative study of programming paradigms, which dealt with procedure calls, pointcut-advice, and annotation by the implementation of a sample program. The comparison was in terms of the locality and the implicitness. Hannemann and Kiczales [15] performed study of improving design patterns by AOP, which showed a comparison of Java and AspectJ by implementation of sample programs including design patterns. The comparison was in terms of the modularity and the reusability. Our study is inspired by these studies and focuses on COP. We perform comparative studies in terms of the description amount and the locality of change measured by basic metrics such as LOC and chunks of code as a first step towards the evaluation of the efficacy of COP to deal with CCCs.

6. CONCLUSIONS AND FUTURE WORK

As the programming paradigm that achieves SoC with a different approach from existing paradigms, COP has been proposed. From the viewpoints of the description amount and the locality of change, we performed comparative studies of OOP, AOP, and COP to detect situations in which COP achieves better SoC and determine the reason why COP can or cannot achieve better SoC. From these results, several avenues for future work exist.

The next step would be a comparative study with larger projects applying metrics used in prior works [11, 13]. A comparative study focusing on multi-perspective analysis would also be interesting, for instance, reusability, implicitness, ease of description, and learning cost. A second line of work would be to discuss and develop a COP language that improves the inferior situations detected in this research.

References

[1] Appeltauer, M. et al. : Event-Specific Software Composition in Context-Oriented Programming, SC 2010, pp. 50-75.

[2] Appeltauer, M. et al. : Context-oriented Programming with Java, 26th JSSST Annual Conference, 2009.

[3] Salvaneschi, G. et al. : JavaCtx: Seamless Toolchain Integration for Context-Oriented Programming, COP'11, 2011.

[4] AspectJ, http://www.eclipse.org/aspectj/

[5] Kiczales, G. et al. : An Overview of AspectJ, ECOOP '01, pp. 327-353, 2001.

[6] JCop-Context-Oriented Programming Projects, https://www.hpi.uni-potsdam.de/hirschfeld/trac/Cop/wiki/JCop

[7] JHotDraw, http://www.jhotdraw.org/

[8] Qt Jambi, http://qt-jabmi.org/

[9] Schippers, H. et al. : An Implementation Substrate for Languages Composing Modularized Crosscutting Concerns, SAC'09, pp. 1944-1951, 2009.

[10] Hirschfeld, R. et al. : Context-oriented Programming, Journal of Object Technology, Vol. 7, No. 3, pp.125-151, 2008.

[11] Figueiredo, E. et al. : Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability, ICSE' 08, pp. 261-270, 2008.

[12] Soares, S. et al. : Implementing Distribution and Persistece Aspects with AspectJ, OOPSLA'02, pp. 174-190, 2002.

[13] Figueiredo, E. et al. : On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework, CSMR 2008, pp. 183-192, 2008

[14] Kiczales, G. and Mezini, M.: Separation of Concerns with Procedures, Annotations, Advice and Pointcuts, ECOOP 2005, pp. 195-213, 2004.

[15] Hannemann, J. and Kiczales, G.: Design Pattern Implementation in Java and AspectJ, OOPSLA'02, 2002.