# A Comparison of Approaches Toward Reusable Aspects

Shun Yi
School of Computer Science and Engineering. Wuhan Institute of Technology
Hubei Province Key Laboratory of Intelligent Robot
Wuhan, China

Chengwan He
School of Computer Science and Engineering. Wuhan Institute of Technology
Hubei Province Key Laboratory of Intelligent Robot
Wuhan, China

***Abstract*—Aspect Oriented Programming (AOP) is designed to reduce maintenance and improve reuse of software components by separating crosscutting concerns from core concerns. In current most mature and widely used AOP approach, such as AspectJ, crosscutting concerns are modularized in aspects and connected to main program using pointcuts. However, the way that aspects and main program are associated leads to tightly coupling between aspects and the target classes and thus the aspects may not be reusable as might be expected. This paper introduces four different AOP approaches that have achieved reusable aspects to a certain extent and makes a comparison of them on four different criteria. Additionally, we put forward two future research directions for better reusable aspects.**

***Keywords—AOP; reusable aspect; comparison; software reuse***

## I. INTRODUCTION

AOP (aspect-oriented programming) has realized the separation of concerns which is hard for traditional programming to achieve [1]. AspectJ [11] is one of the most mature and widely used AOP approach currently. In AspectJ, aspect is used to modularize crosscutting concerns, the pointcuts in aspect capture the execution points of the target program called join points, and the advices in aspect defines what behavior to be added at the join points. Finally, a weaver will weave the advices into the target program. These aspects are defined on the basis of features of the target program, it leads to a tight and fragile coupling between aspect and target program, and it reduces the reusability and maintainability of aspects vastly. Additionally, fine-grained configurability and variability have not been supported and realized in current AOP approaches like AspectJ and thus it is very hard to reuse aspects in different contexts [4].

The main purpose of AOP is to reduce the complexity of the program development process, aspects should be reusable in different contexts among one application or in several applications. Thus, how to design reusable aspects that are not coupled with a particular application has arose more and more attention. In this paper, we introduce four AOP approaches that have achieved reusable aspect to a certain extent, including: Caesar [8], ParaAJ [2], JAsCo [10] and Framed Aspects [4]. We mainly focus on how the aspects are reused and make a comparison of the four approaches. They are compared on four different criteria: whether the approach is lightweight or not, configurable or not, language independent or not, support generative aspects or not.

Section Ⅱ provides the comparison criteria. Section Ⅲ introduces the selected approach and compare them using our criteria. Section Ⅳ puts forward some future research directions and section Ⅴ concludes the paper.

## II. COMPARISON CRITERIA

In order to assess these approaches, we put forward four criteria: light weight, configurability, language independence and generative aspects. Here we provide definitions of these criteria.

Light weight: AOP approaches can be divided into two varieties: heavyweight AOP and lightweight AOP. Heavyweight AOP has the aspect-oriented concept itself, like AspectJ. On the other hand, lightweight AOP is preferred because it is based on existing object-oriented language and only extra configuration files are needed to describe the basic information of Aspects. The aspects will be weaved into unaltered language such as Java and C#.

Configurability: when an aspect is reused in different context, the weaver should be able to apply the aspect to target program according to different settings. The configurability of aspect mainly includes: definition of pointcuts, assignment of advice and coordinating the relationship between different aspects.

Language independence: the idea of AOP is essentially language independent. But, most current AOP approaches are depended on a particular language and it can limits the range of aspects reusability.

Generative aspects: generating aspects automatically during AOP programming can improve the efficiency of aspect reusability.

## III. APPROACHES ANALYSIS

### *A. Caesar*

The most important notion of Caesar [8] proposed by Mira Mezini and Klaus Ostermann is the aspect collaboration interface (ACI) which defines aspect as a set of mutually recursive abstractions that interact via well-defined interfaces and it provides better modularity on top of join point interception(JPI) mechanism offered by AspectJ. An ACI mainly consist of several mutually recursive nested ACIs and every ACI is mapped to an abstraction in the modular structure of an aspect.

ACI is designed to decouple aspect bindings from aspect implementations, and define them in individual but indirectly connected modules as they implement disjoint parts of a common ACI. In order to gain a complete realization of an aspect, an implementation-binding pair needs to be composed into a new unit called a *weavelet* [8] which is a new class within it the respective implementations of the methods from the binding and implementation parts are composed. In this way, implementations and bindings can be inherited independently and reused flexibly by a deployment aspect.

In the Caesar model, Java classes with some additional features are used, and it is an extension of Java, thus it is a heavyweight AOP approach and depends on a particular language. There are mainly two improvements made by Caesar model compared with traditional AOP approaches (AspectJ):

- The description of join points in target class are implicit in AspectJ, while it is explicit in Caesar. It improves developer's expressiveness as they can choose among several constructors of the binding classes leading to better reusability.
- The implementation-binding pair of an aspect must be deployed explicitly in Caesar, so that it does not have any effect on the base application's semantics when compiling a binding class that contains advice definitions.

Unfortunately, Caesar can't generate aspects automatically and it does not support configurability of aspect.

## B. *ParaAJ*

In traditional AOP language, such as AspectJ, the association between target class and aspect is class-directional association [15], namely the definition of aspect is under the condition of fully knowledge of the target class, while the target class is oblivious to the modification of their behavior made by aspects. So that we can effectively separate the crosscutting concerns and the main program, remaining clean and reusable classes. But, in this kind of association, an aspect contains the crosscutting logic and target class information at the same time, this will lead to tight coupling between aspects and the target class, making the reuse of aspects difficult. This kind of coupling is due to the lack of consistent interface between target class and aspect [14].

ParaAJ [2] proposed by Khalid Aljasser and Peter Schachte is an extension of the AspectJ. In ParaAJ, aspects are defined with parameters called Parametric Aspects (para-aspect) [2]. A para-aspect can be declared with the following syntax:

```
<visibility>  aspect  <name>  (<formals>)  {
        <aspect body declarations>
}
```

Fig. 1. Para-aspect syntax

In Fig. 1, <formals> specifies the formal parameters of the aspect. Parameters for aspects are similar to the parameters of general methods, each parameter statement consists of parameter type and a parameter name. The meta-type can be any primitive Java type, or String, or one of the following: any Java identifier; a primitive type, class name, or interface name; a field in the target class; a method in the target class; any valid pointcut.

A parametric aspect need to be applied to a specific target class through the "apply" statement with the parameters specified. An aspect can be applied to multiple target classes, and a target class can be repeatedly associated with the same aspect with different parameters. There are two usages of "apply" statement:

*1) Internally:* the "apply" statements can be used in the target class. This kind of association is aspect-directional association, namely aspects can be refered in target class and aspects are oblivious to the target class.

*2) Externally:* the "apply" statements can be used in another class called Host. This kind of association is closed association, namely aspects and target classes are oblivious to each other.

Every "apply" statement create a separate instance of aspect and only affects the target class that it is applied to. ParaAJ uses improved Aspect Bench Compiler (ABC) [16] as a compiler, the compiler will automatically recognize the target class from the "apply" statement, so that the pointcut model in ParaAJ does not contain any information of specific target class, which makes aspects more reusable.

There are mainly two improvements made by ParaAJ compared with traditional AOP approaches (AspectJ):

- Aspects must be applied explicitly to the target class it influence.
- Developers may use parameters to specify how the aspects are applied to application when reused.

The parameters between the aspects and the target class formed a formal interface, so that they can be developed independently, leading to better reusability of aspects. ParaAJ is an expansion of AspectJ and relies on the Java, thus it is a heavyweight AOP and not language independent. ParaAJ implements the configurability through parametric aspect, but it does not support automatic aspect generation.

## C. *JAsCo*

JAsCo is a new AOP approach proposed by Davy Suvée, Wim Vanderperren and Viviane Jonckers which combines the idea of aspect-oriented software development (AOSD) and component based software development (CBSD). JAsCo combines the join point expression in AspectJ and the thought of aspects independent in Aspectual Components [9]. JAsCo introduces two new concepts: aspect beans and connectors, and stays as close as possible to the regular Java syntax.

Aspect bean is a Java bean that contains one or more logical related inner class inside. This inner class is a generic reusable entity called *Hook*, and it may be regarded as combination of abstract pointcut and advice in AspectJ. In *Hook*, abstract pointcuts can be defined as the constructors of *Hook*, and advice can be defined in the methods of *Hook*. The definition of aspect bean is independent from specific context, so that it can be reused in different components. Aspects are deployed by a generic entity called *Connector*. Abstract pointcuts are bound to concrete pointcuts through specifying the constructor's parameter with concrete method signature when Hooks are instantiated in connectors, so that the crosscut logic is bound to a specific application. In the Connector, crosscut logic will be executed when Hook instances call their methods. Crosscut logic can be reused by every instance of Hooks.

JAsCo is an extension of Java with additional features, so it is a heavyweight AOP approach. In JAsCo, the constructor in Hook supports override, so that Hook can be instantiated with different parameters and applied to different target class, which improves reusability and configurability of aspects. Additionally, JAsCo supports a good combination strategy, priority control mechanism of aspects, and dynamic aspect application and removal, and all these features contribute to the flexibility and configurability of aspect reuse. However, compared with general static language (such as AspectJ), the dynamicity and flexibility of JAsCo demands larger performance overhead, thus JAsCo is not applicable in the case of insufficient resources. Currently JAsCo depends on Java language, and cannot be applied to application implemented by other languages. And it does not support the automatic aspect generation.

### *D. Framed Aspects*

AOP approaches such as AspectJ do not allow the separation between specification for a concern and the aspect itself. The aspect is, therefore, a white box component and an intricate understanding of the aspect code is required. Thus, we cannot reuse aspects in a black box manner and the potential for aspects to be reused in different contexts is limited.

The framed aspect [4] proposed by Neil Loughran and Awais Rashid supports reusable aspects by combining the respective strengths of AOP and frame technology [3]. In this approach, AOP is used to modularize crosscutting and tangled concerns and frame technology is used to allow aspects to be parameterized and configured to different requirements. Framed Aspects support parameterization for AOP which enables aspects to be customized in different contexts and thus increases the reusability of an aspect module. Framed aspects consist of three distinct modules: the framed aspect code (normal and parameterized aspect code), composition rules (possible legal aspect feature compositions, combinations, constraints and controls how these are bound together), and specifications (developer's customization specifications). These modules are composed to generate customized aspect code using a frame processor. Framed aspects achieve aspect dependencies through composition rules which also contribute to the reusability of an aspect.

Although the framed aspect mainly focus on one particular AOP approach (Aspect J), but the concepts and meta language are generic and thus it can be applied to other AOP approaches. The frame processor in this approach is actually a simplified version of the XVCL [13] frame processor which can provide effective parameterization and reconfiguration support for aspects. Moreover, in this approach, customized aspect can be generated automatically by bounding the specification, compositions rules and framed aspects together which contribute to the efficiency of aspect reusability.

TABLE I below shows the results of the above discussion and summarize the comparison. "+" means that the approach fulfilled the comparison criteria, and "-" means otherwise.

TABLE I. SUMMARY OF COMPARISON

| **approaches** | **Comparison Criteria** | | | |
|---|---|---|---|---|
| | ***Light Weight*** | ***Configurabi lity*** | ***Language Independence*** | ***Generative aspects*** |
| Caesar | - | - | + | - |
| ParaAJ | - | + | - | - |
| JAsCo | - | + | - | - |
| Framed Aspects | + | + | + | + |

## IV. FUTURE WORK

The approaches compared above has achieved reusable aspect to a certain extent, but it still has room for improvement. In order to achieve better aspect reuse, this section puts forward two research directions for better reusable aspects.

*1) Combination of AOP and software reuse method:* Just as JAsCo approache combines the idea of aspect-oriented software development (AOSD) and component based software development (CBSD), many more software reuse methods in traditional OO programming could be introduced into AOP. The combination of AOP and software reuse method may gain the advantages of both, leading to more reusable aspect and reducing the tangling and scattering codes in traditional programming. In the future, we may try to research the possibility to combine other software reuse method with AOP.

*2) Generic reusable aspect library:* a reusable aspect library that can be easily applied to a variety of applications will truly improve AOP to a new level. Currently some recurring crosscutting concerns like: security[5], concurrency[6][7], distribution[17] and failures[18] have been capsulated into practical aspect libraries. These libraries are all designed to solve a particular problem and depend on a particular language. The common model for a generic aspect library that is AOP platform independent is still missing. The generic aspect library is expected to be language independent and able to organize reusable aspects in a efficient way. In the future, we may do research on architectural patterns and design patterns for a generic reusable aspect library.

## V. CONCLUSION

This paper introduce four AOP approaches that has achieved reusable aspect to a certain extent, including: Caesar, ParaAJ, JAsCo and Framed Aspects. We mainly analyzed how the aspects are reused and make a comparison of the four approaches. The main task to achieve reusable aspects is decoupling aspects from main program. The approaches mentioned above give us many inspirations. From the comparison analysis, it can be seen that lightweight, configurable, language independent and automatic aspect generation are outstanding feature of AOP approaches that can achieve better aspect reuse. We believe that a combination of AOP and software reuse method and a generic reusable aspect library will contribute a lot to aspect reuse.

## ACKNOWLEDGMENT

This research project was supported by National Natural Science Foundation of China under Grant No.61272115, 60873024, 61402150, the Key science Technology Research Project of Hubei Provincial Department of Education under Grant No.D20121508.

## REFERENCES

[1] Maarten Bynens, Eddy Truyen and Wouter Joosen, “A System of Patterns for the Design of Reusable Aspect Libraries,” T. Aspect-Oriented Software Development 01/2011; 8:46-107.

[2] Khalid Aljasser, Peter Schachte, “ParaAJ toward Reusable and Maintainable Aspect Oriented Programs,” Thirty-Second Australasian Computer Science Conference (ACSC 2009), Wellington, New Zealand, January 19-23, 2009.

[3] Bassett, P., “Framing Software Reuse: Lessons from the Real World,” Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1997).

[4] Loughran, N., Rashid, A. “Framed aspects : Supporting variability and configurability for aop,” In: International Conference on Software Reuse (ICSR-8), Springer Berlin/Heidelberg (2004) 127–140.

[5] Huang, M., Wang, C., Zhang, L. “Toward a reusable and generic security aspect library,” In De Win, B., Shah, V., Joosen, W., Bodkin, R., eds.: AOSDSEC: AOSD Technology for Application-Level Security, 2004.

[6] Soares, S., Borba, P. “Implementing Modular and Reusable Aspect-Oriented Concurrency Control with AspectJ,” In: WASP 2005, Uberlandja, Brazil (2005).

[7] Carlos A. Cunha, João Luís Sobral,Miguel P. Monteiro. “Reusable Aspect-Oriented Implementations of Concurrency Patterns and Mechanisms,” Proceedings of the 5th International Conference on Aspect-Oriented Software Development, AOSD 2006, Bonn, Germany, March 20-24, 2006.

[8] Ostermann, K., Mezini, M. “Conquering aspects with Caesar,” Akşit, M., ed.: Proc. 2nd Int’ Conf. on Aspect-Oriented Software Development (AOSD-2003), ACM Press (2003) 90–99.

[9] Lieberherr, K., Lorenz, D., & Mezini, M. “Programming with Aspectual Components,” Technical Report, NU-CCS-99-01(March, 1999).

[10] Suvée, D., Vanderperren, W. “JAsCo: An aspect-oriented approach tailored for component based software development,” Akşit, M., ed.: Proc. 2nd Int’ Conf. on Aspect-Oriented Software Development(AOSD-2003), ACM Press (2003) 21–29.

[11] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. & Griswold, W. G. (2001), “An overview of AspectJ,” Lecture Notes in Computer Science 2072, 327-355.

[12] M. Mezini and K. Ostermann. “Integrating independent components with on-demand remodularization,” In Proceedings of OOPSLA ’02, 2002.

[13] XVCL homepage, http://fxvcl.sourceforge.net

[14] Filman, R. E. & Friedman, D. P. (2000), “Aspect-oriented programming is quantification and obliviousness,” in ‘OOPSLA 2000 Workshop on Advanced Separation of Concerns’, Minneapolis, MN.

[15] Kersten, M. & Murphy, G. C. (1999), “Atlas: a case study in building a Web-based learning environment using aspect-oriented programming,” ACM SIGPLAN Notices 34(10), 340-352.

[16] Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G. & Tibble, J. (2005), “abc: An extensible AspectJ compiler,” Transactions on Aspect-Oriented Software Development.

[17] Soares, S., Laureano, E., Borba, P. “Implementing distribution and persistence aspects with AspectJ,” In: Proceedings of OOPSLA, ACM Press (2002) 174–190.

[18] Kienzle, J., Guerraoui, R. “AOP - Does It Make Sense? The Case of Concurrency and Failures,” In Magnusson, B., ed.: 16th European Conference on Object–Oriented Programming – ECOOP 2002. Number 2374 in Lecture Notes in Computer Science, Malaga, Spain, Springer Verlag (2002) 37 – 61.