Aspect Oriented Programming and Composition Filters: A Conceptual Comparative Study

Abdel Hakim Hannousse, Djamel Meslati, Hayette Merouani

LRI Laboratory, Badji Mokhtar University of Annaba, BP 12, Annaba, Algeria, 23000
hannousse_a_hakim@yahoo.fr

Abstract. The Object Oriented Model has some limitations that recent approaches known as Advanced Separation of Concerns ASOC try to eliminate. Today, there are many ASOC approaches and their comparison is increasingly considered as an important issue. Unfortunately, few works are dedicated to the comparison and the assessment of these approaches. In this paper, we present an assessment of two ASOC approaches: Composition Filters (CF) and Aspect Oriented Programming (AOP) through a conceptual comparative study. To achieve this goal, we have performed a translation of the first one to the other. Our work consists of testing the two approaches by confronting their concepts. The translation carried out, proved its efficiency to be helpful and our conclusions show that the mapping of concepts is not straightforward or one-to-one. To make our approach significant, we have implemented the CoAspectJ preprocessor that accomplishes our mapping model automatically to validate our rules.

1 Introduction

Some of recurrent problems captured in object oriented software design are code tangling and code scattering problems. These both problems affect the development process of the application in different manners: bad traceability, lack productivity, weak code reusability and quality and difficult application evolution. To avoid these problems, several techniques are being researched that attempt to increase the expressiveness of the OO paradigm. Such techniques are known as Advanced Separation Of Concerns (ASOC).

Currently, a large amount of literature is devoted to three advanced separation of concerns: Kiczales Aspect Oriented Programming (AOP) [12], Composition Filters (CF) [1], and Multi-dimensional separation of concerns [20]. All these approaches aim to providing better concepts and mechanisms to make sure that different concerns are represented by different modules in a program. Unfortunately, their philosophies are different from others and their concepts used are not similar. In front of this diversity, software community hopes to unify concerns modularization and increase software reuse, evolution and maintainability in a standard manner [25].

In order to contribute for achieving this goal we have undertaken a conceptual comparative study. Our idea consists of using AOP as a pivot approach to which we compare the other approaches through mapping concepts. This choice is due to the

fact that AOP has now reached a certain maturity and a lot of developers and programmers are using it [9, 22, 23]. The work presented here is limited to a conceptual comparison of the AOP and the CF approaches.

The remainder of this paper is organized as follows: Section 2 and 3 describe the main concepts of AOP and CF respectively in which based our comparative study. Section 4 shows how AOP concepts are mapped into CF. Section 5 and 6 present our results obtained from the mapping section. Section 7 discusses related work and section 8 summarizes our work.

2 Aspects Oriented Programming

Aspect Oriented Programming or AOP in short considers that the code of a given software system contains a functional concerns and a non-functional concerns. The functional concerns are concerns of system that achieves basic functionalities of the application. Non functional concerns represent each other concerns that crosscut basic functionalities. In the object-oriented approach, concerns are scattered throughout the source code, what creates an entanglement [12]. So, the vision motivating AOP is that one could provide independent specifications for each concern or aspect and then weave them together to build the resulting system.

AspectJ is a general-purpose AOP extension to Java language. It allows the definition of concerns that are called aspects. A weaver applies aspect definitions over source code (Java classes) and creates the woven version, also interpreted by the Java Virtual Machine (JVM). AspectJ extends the Java language with five new concepts: Aspects, Join points, Pointcuts, Advices and Inter-type members' declaration or introductions:

1. **Aspects:** An aspect is an entity that looks like a class but model a concern that crosscut several object classes. Aspects are defined one by one. Several aspects can exist in the same software system.
2. **Join points:** Are well-defined places in the structure or execution flow of a program where additional behavior is attached and they constitute basic code breaking points where aspect can get involved. Those points are basically: object instantiations, method invocations or executions, field setters and getters, exception handlers**,** etc.
3. **Pointcuts:** They are particular forms of predicates that use Boolean operators and specific primitives to pick out join points and dynamic contextual information.
4. **Advices/actions:** The specified events can be caught, and actions can be applied to take care of them. Those actions are called advices. They are method-like mechanisms used to declare that a certain code should execute at each of the join point in a pointcut. So, the code of advice runs at every join point picked out by its pointcut. Exactly how the code runs depends on the kind of advice. AspectJ supports three kinds of advice. The kind of advice determines how it interacts with the join points which defined over. Thus, AspectJ divides advices into that which runs before its join points, that which runs after its join points, and that which runs in place of (or "around") its join points.

5. **Introductions:** It enables the specification of static crosscutting of the functional code by adding members to classes or by specifying what a class extends or implements. So, desired code can be added to existing applications at the level of class definitions, such as, methods, instance variables and inheritance structure.

To understand the philosophy of this implementation model we can imagine a system during its execution where the system is picked up at each moment. The concern specification materialized by one or more aspects can interfere at observable points of the execution such as: getting or setting an instance variable, calling or executing a method, throwing an exception, etc. These points also called join points. Interfering a concern consists of executing its appropriate specification materialized by one or more advices.

3 Composition Filters Approach

A fundamental design decision of the CF model is to distinguish two kinds of abstractions: (class-like) concerns and filters. Briefly, a concern is the unit for defining the primary behavior, while filters are used to extend or enhance concerns so that (crosscutting) propriety can be represented more effectively. So, CF adds a wrapping layer called *interface* to the conventional object model that intercepts incoming and outgoing messages. The main components of the interface layer are:

1. **Internal objects:** They are objects whose methods are used to compose the behavior of the CF object. Messages received by a CF object can be delegated to the internal objects instead of the kernel object. Internal objects are encapsulated in the CF object and do not exist beyond its existence.
2. **External objects:** They are almost like internal objects. However, they are supposed to exist on their own and their references are passed on to the constructor of the CF class during the instantiation. These references are assigned to the CF instance variables.
3. **Methods:** The interface declares all methods available to other objects.
4. **Conditions:** Conditions are specific methods that inform about the state of the kernel. They do not have parameters and do not affect the state of the kernel.
5. **Input filters:** A set of declarative specifications that intercept the incoming messages.
6. **Output filters:** A set of declarative specifications that intercept the outgoing messages.

We must denote here, that filters are declared in ordered sets. So, a call entering to a CF object is first reified then passed along the filter set until it's discarded or dispatched. The existence of many filter types makes this approach more significant. Each filter interface can materialize a concern, where more than one concern can affect the software system, thus, more than one filter interface can be applied to the same object but in ordered set.

In order to add crosscutting concerns (i.e. methods, conditions and filters) to the one or more objects, the composition filters model provides the superimposition mechanism. Superimposition is expressed by a superimposition specification, which specifies how and when the concerns crosscut each other.

4 The Concepts Mapping

The idea behind the mapping consists of answering to the question: For each given specification of one of the approaches, what is the corresponding specification in the other? In our work, we try to identify the correspondence between both AOP and CF approaches.

A system in CF approach corresponds to a set of units each one consists of a class extended by interface specification. Let CFU be a CF unit:

CFU = <*K, I*> where:
K: the kernel part.
I: the corresponding interface
K = <*M, IV*> where:
M: public methods set
IV: instance variables set.
I = < *F, Externals, Internals, MC, SP* > where
F: is filter set,
Externals: is a set of instance variables containing references of external objects,
Internals: is a set of instance variables containing references of internal objects,
MC: is a set of conditions and methods
SP: is the superimposition specification

The counterpart of *CFU* in AOP is < *K, A*> where:

K: is a like K in CFU that represents a set of classes they implement the functional pat of a system. It contains the same set of methods and instance variables in CFU.
A: a set of aspects that represent the non-functional part.

Since *I* materializes several concern parts, its translation will be composed of several aspects "A": An aspect for each filter. The *interface* aspect is used to introduce Externals, *Internals* found in *I* to the kernel. Each public method in *I* has an empty implementation body in the interface aspect, since the corresponding calls will be picked out by aspects and delegated to internals or externals. Internals and externals classes themselves remain unchanged.

So, we can represent A formally by:

A = <*IA, FA*> where:

IA: represents an aspect contains declarations of internals, externals, instance variables and an introduction of empty methods correspond to public methods contained in internals and externals.

FA=<AA, CA> where:

AA: is a set of abstract aspects each one contains a specification of semantic of one filter type, so, they number is the number of filter type in CF model.

CA: is a set of a concrete aspects each one must inherit from the equivalent filter type (one of AA set), so, each CA element match a filter in the F.

```
aspect Interface {
   kernel.Internals = new Internals();
   kernel.Externals;
   public type kernel.publicMethod(..) {}
}
```

Each predefined abstract aspect implements one filter type, so, it contains two predefined methods: accept and reject methods each one implement the accepting and the rejecting actions of the filter type respectively.

```
abstract aspect Filter_filterType {
   public void accept() { // accepting action }
   public void reject() { // rejection action }
}

aspect idFilteri extends Filter_filterType {
   pointcut
idFilteri_accept():if(ConditionPart)&&if(MatchingPart)
   before() : idFilteri_accept() {
          accept();
   }
   before() : !idFilteri_accept(){
          reject();
   }

}
```

Each filter element (FE) has a corresponding *filter_accept* pointcut in our mapping model. This pointcut is divided into two parts condition part and matching part, the first one is used to evaluate the condition part; so, it's translated using a user-defined primitive pointcuts that contain the same expression of the corresponding condition part. In case where the condition part includes a specification of a pseudo variable we use the reflection level of the corresponding software to determine it. The second part contains the specification correspondent to the matching part of the filter element. This later is translated using the reflection level specified by a specific method match in the predefined matching aspect.

The following table presents the summary of our mapping model of the two approaches based in their main concepts.

Table 1. Our mapping model used to the assessment of AOP and CF

Aspect Oriented Programming	Composition Filters
Functional code	The Kernel part of the CF object model
Non-functional code	Interface part of the CF object model
Aspect	Filter
Aspect name	Filter name
Abstract aspect	Filter Type
Pointcut	Filter element
Condition pointcut	Condition part of filters
Pointcut specification	Matching part
Aspects precedence	Filters order in the filter interface part
Advice	Filter Semantic (i.e. acceptance and rejection)
Weaving specification	Superimposition specification
Private members	Encapsulation of objects

5 Some Challenging

Some challenging have encountered during the mapping stage, most of them due to the CF semantic of messages processing:

1. Passing messages through filters and evaluating the filter elements is done in a specific order corresponding to their declaration in CF model: Up-down for filter set and from left to right for filter elements.
2. In case of a final decision made by a filter corresponding to a message, the remaining elements in the same filter and the followed filters won't be considered for the current message.

To reflect the first point, we specify the dominance between aspects. The aspect corresponding to the first filter will dominate the aspect corresponding to the second that will dominate the aspect corresponding to the third and so on. The filter elements order is enforced by declaring the corresponding advices in the same order of the filter elements.

To reflect the second point, it is necessary to inhibit advices and aspects corresponding to the remaining filter elements and filters. To attain this purpose, we have added a Boolean instance variable initially evaluated to true, this instance must be evaluated first, if it's evaluated to true and the current message accepts the condition part and matches the matching part then the instance must be enforced by assigning it the false value to inhibit the handling of the current message by the followed filter elements in the same aspect and in the other, then it apply their corresponding advices. In addition, the last filter element must evaluate the Boolean instance value to true to initialize the system.

6 Assessments and Comparison

The originality of our work resides in putting the approaches in a test stage by confronting their concepts together. The translation carried out, proved its efficiency to be helpful and allowed us to determine the main features, differences and insufficiencies of approaches. We present them in what follows with some practical results.

6.1 Main Features

Aspect Oriented Programming aims to separating a system to concerns which represent either functional or a non functional parts. Therefore, we can characterize each part independently and their mutual interactions using the two approaches.

Features of the functional part: The CF like AOP allows expressing their functional part in terms of conventional object model and preserving its independency from the non-functional part.
Features of the non-functional part: CF describes its non functional part using a simple declarative style with a clear semantics by specifying filter interfaces. That makes CF more independent from every existing paradigm such as Object Oriented Programming; however, that increases the expressiveness of the approach. In contrast, AOP implementation model uses a specific mechanism that is strongly oriented toward the paradigm used in the functional part and that makes the code more expressive and easy to understand.
Features of interactions between functional and the non-functional parts: The functional part is that generates states and events that trigger the concerns. AOP is provided with a wealthy join points model that allows expressing various concerns in different ways. In opposite, the CF interaction is no more than messages intercepting. So, the interaction part is richer in AOP than in CF model: It is possible to wrap interesting points within methods such as variable sets and gets and then intercept calls to these methods in AOP. Unfortunately, this cannot be done without altering the functional code classes in CF.

Interactions between the functional and the non functional part signify that the first one must be altered by the second; the interaction mechanism can be expressed in terms of behavior substitution and behavior extending:

Behavior substitution: It means that substituting a behavior by another that may belong to another object. In this case, we use the term *delegation*. The CF expresses this concept directly by providing the substitution part in the filter elements specification. AOP provides around advice to freely control calls: resuming or substituting them.
Behavior extending: Consist in adding pieces of code to the original in a given point of the execution. AOP allows additions by the three kinds of advices. In the CF, the addition can take place by either by intercepting a call and using the filter *Meta*, that can be used to substitute the called method by a method of an internal object that adds

the necessary behavior and resumes the original call, or by using selectors and methods parts used in the superimposition specification.

The (Un) pluggability of the non functional part: The (un)pluggability of the non functional part is one of the important features that must be considered in many real applications at runtime. The AOP approach with all its implementation models is statically based approach, where activating or deactivating some concerns must be done in statically mode (at compile time), whereas, the CF approach presents the advantage that it can be used at runtime such as Sina/ST model or compiletime such as ComposeJ or ConcernJ model. So, the CF is more practiced than the AOP approach.

6.2 Practical Results

Currently, we have achieved a CoAspectJ preprocessor that accepts as input CF programs specified in ConcernJ and translates them in corresponding ASPECTJ programs. The present version does not operate any optimization but allows us to show that our translation rules are correct. Notice that those practical limitations did not have negative influence on our work since our focus is on the conceptual comparison. In some practical examples used with our preprocessor, we noticed that the concerns generated code has a longer size and is more difficult to understand than the original code.

7 Related Work

Because AOP and CF are new paradigms, compared to OOP, and since they are continually evolving, only a limited amount of research work is devoted to the assessment in general and the performance issues in particular. In the same time, this type of research work is necessary to guide the evolution of the two approaches towards promising issues. We classify current works in three groups:

1. Assessment through practical use [26, 18]. In this type of work, the objective is to get a subjective assessment of concepts when used by programmers and/or quantifying efforts and time necessary to implement an AOP application through case studies.
2. Assessment through implementation of particular applications like design patterns, exception handling, distribution, etc [8, 9, 16, 17, 19, 22]. Here the purpose is to show how well emerging AOP approaches tackle subtle problems.
3. Implementation works. This covers works dedicated to the implementation of AOP concepts and weaving techniques [10, 21].

Our work is complementary to all these works. Since one cannot expect to find out significant insufficiencies while using emerging approaches in situations foreseen by their authors, implementing AOP languages using these approaches becomes an interesting issue and a challenging task. That is what makes our work original.

8 Conclusions

The mapping between AOP and CF approaches shows its efficiency. The conclusion derived from this comparison shows that, even though each approach does not constitute a 'killer application' for the other, neither subsumes the other. Indeed, beyond the common features we noted differences and insufficiencies. Differences preclude a straightforward and one-to-one mapping, whereas insufficiencies lead to a translation where concerns generated code is tangled. For this last point, we put the stress on the fact that the two approaches lack a suitable control of concerns needed to enforce each other semantics. To improve the concern control in the CF, we proposed a solution consisting of introducing sequences of substitution and a certain handling of parameters. Moreover, we noticed that the translation of the whole AOP in CF is not possible without altering functional code classes. It is the case for example of accessing classes' private members and picking out variables access and assignment join points.

Perspectives of this work are numerous and aim to contributing to the emergence of the unified concern modeling approach. Concepts mapping with other Advanced Separation of Concerns such as multidimensional separation of concerns and Adaptive Programming approach is considered one of our interest.

References

1. Aksit M., Tekinerdogan B., Aspect-Oriented Programming Using Composition Filters, ECOOP'98 Workshop Reader, Springer Verlag, pp. 435, July 1998. http://trese.cs.utwente.nl/composition_filters/
2. Aksit M., Tekinerdogan B., Solving the Modeling Problems of Object-Oriented Languages by Composing Multiple Aspects Using Composition Filters, AOP'98 workshop position paper, 1998. http://trese.cs.utwente.nl/composition_filters/
3. Aksit M., Bergmans L., Software Evolution Problems in Case of Inheritance and Aggregation Based Reuse, Tutorial of the Trese Group, http://trese.cs.utwente.nl
4. Bennett K.H., Rajlich V.T., Software Maintenance and Evolution: A Roadmap, in Anthony Finkelstein (Ed.), The Future of Software Engineering, ACM Press 2000, pp. 73-87
5. Bergmans L., Aksit M., Composing Crosscutting Concerns Using Composition Filters, Communications of the ACM, Vol. 44, No. 10, pp. 51-57, October 2001.
6. Bergmans L., The Composition Filters Object Model, Dept. of Computer Science, University of Twente, 1994. http://trese.cs.utwente.nl/composition_filters/
7. Glandrup M., Extending C++ using the concepts of Composition Filters, MSc. thesis, Dept. of Computer Science, University of Twente, 1995. http://trese.cs.utwente.nl/composition_filters/
8. Gray Jeff, et al, Handling Crosscutting Constraints in Domain-Specific Modeling, CACM, vol. 44, no. 10, pp 87-93, October 2001
9. Hannemann J., Kiczales G., Design Pattern Implementations in JAVA and ASPECTJ. In Proc. of OOPSLA, ACM, (2002).
10. Hilsdale E., Hugunin J., Advice Weaving in ASPECTJ. Submitted to the *3rd International Conference on Aspect-Oriented Software Development (AOSD)*. April 2004. http://www.cs.indiana.edu/~ehilsdal/cv.2004-aosd-adviceweaving.pdf
11. HYPER/J web site: http://www.research.ibm.com/hyperspace/

12. Kiczales G. et al., Aspect Oriented Programming. In Proc. of European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science Vol. 1241, pp. 220-242, 1997. http://eclipse.org/ASPECTJ
13. Kiczales G. et al, An Overview of ASPECTJ In Proc. of ECOOP, Springer-Verlag (2001).
14. Koopmans P., On the design and realization of the Sina compiler, MSc. thesis, Dept. of Computer Science, Univ. of Twente, 1995.
15. Lehman M. M., Ramil, J.F., Rules and Tools for Software Evolution Planning and Management. Annals of Software Engineering 11(1): 15-44 (2001)
16. Lippert M., Lopes C.V., A study on Exception Detection and Handling Using Aspect-Oriented Programming, Proceedings of the 22nd international conference on Software engineering, Limerick, Ireland, ACM Press, pp. 418 – 427, 2000
17. Mendhekar A., et al., RG: A Case Study for Aspect-Oriented Programming, Xerox PARC, Palo Alto, CA. Technical report SPL97-009 P9710044, February, 1997.
18. Murphy G. C. et al, Evaluating emerging software development technologies: Lessons learned from assessing aspect-oriented programming, IEEE Transactions on Software Engineering, pp. 438-455, July/August 1999.
19. Natsuko N., Tomoji K., Implementing Design Patterns Using Advanced Separation of Concerns, workshop on Advanced Separation of Concerns in Object Oriented Systems, OOPSLA 2001
20. Ossher H., Tarr P., Multi-Dimensional Separation of Concerns using Hyperspaces. IBM Research Report 21452, April, 1999.
21. Popovici A. et al, Just-In-Time Aspects: Efficient Dynamic Weaving for JAVA, Proc. 2nd Intl Conf. Aspect-Oriented Software Development, March 2003.
22. Rajeev R. et al, A distributed concurrent system with ASPECTJ, ACM SIGAPP Applied Computing Review, Vol. 9 Issue 2, July 2001
23. Rashid A. Weaving Aspects in a Persistent Environment. ACM SIGPLAN Notices. Volume 37,No, pp. 36-44, 2002
24. Tarr P. et al, N degrees of separation: Multi-dimensional separation of concerns. In Proc. of the 21st International Conference on Software Engineering, pp. 107-119, 1999.
25. Walker D. et al, A theory of aspects, Proc. of the eighth ACM SIGPLAN international conference on Functional programming, Volume 38 Issue 9, August 2003
26. Walker R. et al, An Initial Assessment of Aspect-Oriented Programming. Proc. of the 21st International Conference on Software Engineering, pp. 120-130, 1999.
27. Wichman J. C., The Development of a Preprocessor to Facilitate Composition Filters in the JAVA Language, MSc. thesis, Dept. of Computer Science, University of Twente, 1999. http://trese.cs.utwente.nl/composition_filters