BABEŞ-BOLYAI University of Cluj-Napoca
Faculty of Mathematics and Computer Science
Proceedings of the National Symposium ZAC2014
(Zilele Academice Clujene, 2014), p. 11-18

A Comparison of Aspect Oriented Languages

by
Grigoreta S. Cojocar and Adriana M. Guran[1]

Abstract. *Many different aspect oriented languages have been developed since the first presentation of aspect oriented concepts in 1997 by Kiczales et al. In this paper we compare four of the existing languages using different criteria: the aspect oriented concepts implemented, weaving, and whether they require source code modification.*

Keywords: aspect-oriented languages, comparison

1. Introduction

Developing software systems that can be easily evolved, modified, and maintained is one of the main goals of software developers. However, practice has shown that developing such software systems is not easy. One of the problems is the "tyranny of the dominant decomposition" [TOH+99]. A software system is composed of many core concerns and (some) crosscutting concerns. If core concerns can be cleanly separated and implemented using existing programming paradigms (like object oriented paradigm), this is not true for crosscutting concerns, as a crosscutting concern has a more system-wide behavior that cuts across many of the core concerns implementation modules. The aspect-oriented paradigm (AOP) is one of the approaches proposed, so far, for overcoming this prevalent decomposition [KLM+97].

[1] Babeş-Bolyai University, Cluj-Napoca, *grigo/adriana@cs.ubbcluj.ro*

Aspect oriented paradigm is used only for crosscutting concerns, the core concerns are still designed and implemented using the base programming paradigm. For crosscutting concerns development, AOP introduces new concepts: *join point*, *pointcut*, *advice*, *aspect* and *introduction* for the design and implementation, and *weaving* for building the final software system. In the following, these concepts are briefly explained:

- A *join point* is a well-defined point in the execution of a program. Any software system can be seen as a sequence of execution points like: assignments, conditional statements, loop statements, methods calls executions, etc. regardless of the programming paradigm used for developing the system. AOP only uses some of these points, called join points, in order to add new behavior.
- Not all the join points that appear during the execution of a software system are necessary for the design and implementation of crosscutting concerns. A *pointcut* selects the necessary join points, and exposes some of the values in the execution context of these join points.
- A pointcut allows selecting join points from the software system, but they do not change the behavior of the system. An *advice* defines crosscutting behavior and it is defined in terms of pointcuts. The code of an advice runs at every join point selected by its pointcut. There are different options as to when the code of the advice is executed relatively to the corresponding join point(s): `before`, `after` or `around` the join point. For the after advice there can be three situations, depending on the execution of the join point: `after returning` (the advice code is executed only if the join point execution completes normally), `after throwing` (the advice code is executed only if the join point execution ends by throwing an exception), and `after` (`finally`) (the advice code is executed regardless of the means by which the selected join point exits (normal or exceptional return)).

- Sometimes, in order to design and implement a crosscutting concern, it is necessary to modify the static structure of a type (by adding new members - attributes/methods or by modifying its inheritance hierarchy). Even though an advice adds new behavior to existing types, it does not modify their static structure. An *introduction* allows developers to extend the static structure of existing types. New methods and/or attributes can be added, or the type inheritance hierarchy can be modified (by adding new interfaces or by adding a base type to an existing type).
- An *aspect* is a new kind of type that is used to implement one crosscutting concern in a modular way. An aspect is similar to a class, it can contain attributes and methods declarations, but it also encapsulates pointcuts, advice and introductions.
- When AOP is used for developing software systems, the core concerns are developed independently of the crosscutting concerns. However, in the end, they still have to be put together in order to obtain the final executing system. *Weaving* is the process that produces the final system, and the *weaver* is the tool used to obtain it. The weaver takes some representation of the core concerns (source code or binaries), some representation of the crosscutting concerns (source code or binaries) and produces the output, which is often a binary representation.

2. Comparison

2.1 Aspect Oriented Languages

The aspect oriented languages considered for this comparison are: AspectJ for Java, the first aspect-oriented extension developed and the most used both in industry and in the academia [AspectJ], SpringAOP, that is Spring framework implementation of AOP concepts [SpringAOP], PostSharp, an aspect-oriented implementation framework for C# [PostSharp], and AspectC++, an aspect oriented extension for C++ [AspectC++].

2.2 Selected Comparison Criteria

In order to compare the chosen aspect oriented languages we use the following criteria: the aspect-oriented concepts supported (the join points that can be selected, advices, aspects, introductions), the weaving process, and the required core concerns source code modification. For our analysis we have considered three common crosscutting concerns: logging, observer design pattern and security because they require different AOP concepts for their design and implementation:

- *Logging.* In order to implement logging, a new aspect is defined which selects the join points of interest (usually entering a method and exiting a method) and before and after advice for storing the necessary data.
- *Observer*. For the Observer design pattern [GHJ+95] implementation the classes hierarchy is usually changed: one class inherits from the Subject and one or more classes implement the Observer interface [HK02]. The modified classes also need to define new methods for adding/removing observer, and they must provide a definition for the update method.
- *Security*. Different software systems may require different things to be secured: some of them require the execution of certain functionalities to be secured, while others require the data to be secured. Because of that, depending on the software system security type, different join points must be used: getting or setting the value of an attribute, or executing or calling some methods.

2.3 Analysis

In the following we analyze the chosen aspect-oriented languages using the considered criteria: the AOP concepts implemented (the type of join points that can be selected, the kind of advice and of introductions

allowed, how the weaver builds the final system, and how the rules of weaving are specified to the weaver.

2.3.1 Pointcuts. Three of the four aspect-oriented languages (AspectJ, PostSharp, AspectC++) allow the selection of many different join points: method call/execution, constructor call/execution, class initialization, getting/setting the value of a field, handling a thrown exception, etc, but they do not allow selection at statement level: the execution of a for-statement, an if-statement, etc. Spring AOP allows only the selection of methods call/execution and the selection of beans based on their name.

2.3.2 Advice. All the chosen aspect-oriented languages support the three kinds of advice: `before`, `after` and `around`, and the three variations of the after advice: `after (finally)`, `after throwing` and `after returning`.

2.3.3 Introductions. AspectJ and PostSharp allow different kinds of introductions: method and field introductions, base class inheritance, interface implementation, etc. AspectC++ introduces the notion of `slice` that can have attributes and methods. The slice can later be used for modifying the static structure of existing types (adding new fields, methods, changing the inheritance hierarchy). SpringAOP allows only one static modification, which is the introduction of interface implementation to existing types.

2.3.4 Weaving. Depending on the aspect-oriented language**,** the weaving process can take places at different times. AspectJ allows three different times: compile-time, post-compile time and load-time, Spring AOP allows weaving at run-time, PostSharp allows post-compile weaving, while AspectC++ allows compile time weaving. The approach used for weaving also depends on the aspect-oriented language: AspectJ uses byte-code modification, Spring AOP uses dynamic proxies, PostSharp uses intermediate language transformation, while AspectC++ uses source code preprocessing.

2.3.5 Weaving rules specification. One of the premises of AOP is that code corresponding to the implementation of crosscutting concern will not be mixed with the code corresponding to core concerns. It is the responsibility of the weaver to build the final that contains both the core

concerns and crosscutting concerns. In order to build the final system the weaver uses the rules specified by the developers. These rules specify which part of the software system must be modified and how. However, the way these rules are specified depends on the aspect-oriented language.

For AspectJ, the pointcuts and the advice are considered to be weaving rules. The former specifies where and the latter specify how. The pointcuts allow the selection of join points either using the pattern matching of method names, type names, etc. or using annotations. Both criteria have advantages and disadvantages. If names are used for selection, the core concerns do not need to be modified, but every name modification may affect the behavior of the final system, as the new name may not be selected by the selection criteria. If annotations are used, that means that the core concerns code must be modified and recompiled in order to include the annotations, which means that we still have to modify the original source code. There is also the possibility of introducing annotations using aspects, however we still have to specify a selection criterion based on names, and we go back to the first problem.

PostSharp also offers two possibilities: a declarative one or using attributes. The former one does not modify the core concerns source code but it is more difficult to use. The latter one is every easy to use, but it requires the addition of each attribute corresponding to an aspect to all the classes/methods that may be modified by the weaver.

Spring AOP offers the same possibilites as AspectJ, either using name patterns or using annotations.

AspectC++ offers only the possibility based on matching of either method names, type names, etc. so it has the same disadvantages.

Considering the above, not all crosscutting concerns can be easily implemented in all analyzed aspect-oriented languages. For example, we cannot implement security on data level using SpringAOP. Also, the design and implementation of the Observer pattern is more difficult with SpringAOP as it allows only the introduction of interfaces to existing types.

3. Conclusions

In this article we have presented a short comparison of four aspect-oriented languages: AspectJ, Spring AOP, PostSharp and AspectC++. The proposed comparison criteria are: the AOP concepts implemented, weaving, and whether they require core concern code modification. These criteria are important when deciding whether to use AOP for developing a system or not to use it. For example, AspectC++ cannot be used for the development of crosscutting concerns that require availability of core concerns code (such as automatic usability evaluation). Spring AOP cannot be used for crosscutting concerns that need to select different join points like class initialization, or fields getting/setting. AspectJ and PostSharp can be used for different kinds of crosscutting concerns implementation. The latter one is not free, but it has a free express version that can be used for simple crosscutting concerns, but for more complicated ones developers need to buy the full version of PostSharp.

References

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

[AspectC++] AspectC++ homepage, http://www.aspectc.org/.

[PostSharp] PostSharp homepage, http://www.postsharp.net/.

[SpringAOP] Aspect Oriented Programming with Spring, http://docs.spring.io/spring/docs/2.5.4/reference/aop.html.

[KLM+97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, JM Loingtier, J. Irwin. *Aspect-Oriented Programming*. In Proceedings of the European Conference on Object-Oriented Programming, volume LNCS 1241, pages 220 - 242, Springer-Verlag, 1997.

[TOH+99] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. S. Jr. *N Degrees of Separation: Multi-Dimensional Separation of Concerns*. In Proceedings of the 21st International Conference on Software Engineering, pages 107 - 119, May 1999.

[GHJ+95] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley. 1995.

[HK02] J. Hannemann, G. Kiczales. *Design Pattern Implementation in Java and AspectJ*. In Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '02), pages 161 - 173, New York, NY, USA, 2002. ACM Press.