6th International Conference on Applied Informatics
Eger, Hungary, January 27–31, 2004.

Aspect-Oriented Programming On Lisp

Miklós Espák

Department of Information Technology,
University of Debrecen
e-mail: espakm@inf.unideb.hu

Abstract

Aspect-oriented programming (AOP) [1] provides a way to separate crosscutting concerns into distinct modules, usually called aspects. Inspite of the separation, these modules are - of course - not independent from each other. Aspects may have to access and modify some values at an execution point of the base program, and - which is mostly a missing feature of mainstream programming languages - they may require some manipulation of the structure or the behaviour of the base.

Therefore, AOP systems need special support by the language. Unfortunately, the mainstream object-oriented languages (like C++ and Java) lack this support, so implementing an AOP system for them has to be done using extralingual techniques. In the paper I will show the main technical issues of implementing an AOP system. It will be pointed out that Java - which serves as a base for most of the AOP systems - incorporates several disadvantages, which hinder it to be an ideal target of AOP systems. It will also be shown that the Common Lisp Object System (CLOS) provides solutions for all the issues discussed and even more. Implementing an AOP system in CLOS can be done in a natural way, using the tools of the language itself. Even, the system resulted will unify the advantages of the individual Java AOP systems.

Key Words and Phrases: Aspect-oriented programming, CLOS, CLOS MOP

1. Introduction

Separating of concerns is one of the most important objectives during software development. Unfortunately, the boundaries of the individual concerns of a software do not overlap the boundaries of the corresponding program modules in most cases. When a concern cannot be mapped unambiguously to a single program module, it is said to *crosscut* these modules. Aspect-oriented programming (AOP) provides a technique for separating crosscutting concerns into distinct program modules.

In the next section the main technical issues of AOP are discussed. The third section describes the current implementation techniques of AOP systems based on Java and shows that Java suffers from such problems which hinder the implementation to be effective, flexible and comprehensible at the same time. In the fourth section it is shown that Common Lisp provides much better support for implementing an AOP system. I also will give some details of the core of such a system. Finally, I will summarize the advantage of CLOS as the base of an AOP system.

2. Technical issues of AOP

The main purpose of AOP is to provide new kind of program modules for integrating structural and behavioural modifications on some well-defined points of the program structure or flow, respectively. Behavioural changes are specified by *advices*, and structural ones by *introductions*. The program points mentioned are called *join points*. A set of join points can be specified by a *pointcut designator*.[1] Behavioural changes are realized by weaving the body of advices into the program at the join points. Weaving can be carried out at several points during the lifecycle of a program. Accordingly, we speak about compile-time, load-time or run-time AOP systems.

As advices complement or enhance other program parts, they need to be able to extract some metainformation about the current point of the program, and also to refer to the objects at the context of the join point. Representing and/or manipulating metainformation can be done through reflection. In terms of context exposure, the best is if the advices access the context just as if they were physically defined in the same scope as the actual join points. To implement this the language has to provide dynamic scoping or passing by reference.

Structural changes may include introducing new fields into classes or defining methods of existing classes. Sometimes even the change of the inheritance hierarchy is required. Making structural changes on a program from the program itself is called structural reflection.

One of the most important characteristics of an AOP system is the time when weaving aspects into the other part of the application happens. Weaving can be performed essentially at any phase of the life cycle of the program: for example at compile-time, load-time or run-time. Generally, it can be said that weaving at a later phase provides more flexibility but costs more at the same time.

Getting familiar with an AOP system and making more readable programs are better assisted by a system that provides a language, not just an application programming interface (API). Although the API solution is said to be having the advantage of not enforcing the users to learn another language, AO programs written in this way are usually longer, more confusing and more difficult to understand.

As AOP systems are typically based on an already existing language, the support from this language for the issues mentioned above is crucial. Unfortunately,

[1] Some AOP approaches use different terminology.

the support provided by the mainstream object-oriented (OO) languages (like C++ and Java) is very limited, so implementing an AO system on them requires extralingual techniques. Although there are several solutions, they suffer from various problems. In the next section some Java AOP systems will be introduced in brief, in order to point out that the lack of language support can be blamed for their shortcomings.

3. Java AOP systems

There has been a lot of research efforts spent on extending Java to support AOP. As it is almost impossible to discuss all of them in detail, I choose three typical representatives in categories distinguished by the time of weaving.

3.1. Compile-time systems

AspectJ [2] is one of the first AO languages and probably also the most popular one. Its popularity is likely to be explained by the innovative set of language constructs it provides. The use of the language is very intuitive, its programs are easy to read. In order to employ the advantages of language extensions, AspectJ is implemented as a compiler. Unfortunately, beside the positives this technique also has some downsides:

- Weaving is done statically, so unweaving aspects is no more possible at running the program.
- Weaving aspects need the source code of the base program, which makes aspects more difficult to deploy.

Although AspectJ does not provide support for unweaving or disabling aspects, this behaviour can be simulated by defining a simple enabling and disabling method in the aspect. However, this is not equivalent with unweaving, as by disabling an aspect so called hooks remain in the program. Hooks are small pieces of program code inserted into the base program. The role of hooks is to execute advices when needed. When the execution of a hook does not result in invoking any advices, the hook is called empty. Though empty hooks do not affect the behaviour of the program, if in high number they can result in a significant loss of speed. One possible cause of the occurence of empty hooks can be disabled aspects.

3.2. Load-time systems

JAC [3] answers some of the shortcomings of AspectJ. It introduces the concepts of "aspect components". Aspect components are Java classes implementing the `AspectComponent` class. They can be compiled independently from the base application. Aspect components can be parameterized through configuration files. With this technique they are very easy to deploy. Another important difference is

that JAC is a dynamic AOP system, which means that it can weave and unweave aspects at run-time. At the same time, JAC uses static translation: it translates the bytecode of classes statically, just before loading them. Using static translation to implement a dynamic AOP system has serious consequences. First, the set of classes which may be affected by aspects to be woven later must be given before starting the application. This set is fixed, and cannot be modified during the execution. The JAC classloader places hooks at every place of the program which can be a join point later. Although aspect components can be woven dynamically, the set of classes they can be woven into is specified statically: aspects can be woven only into those classes in which hooks were inserted at loading.

Secondly, this way of weaving results in a high number of empty hooks, causing unwanted performance loss in the application. The more flexibility of the application is needed, the wider set of classes has to be specified in the configuration files, so the number of empty hooks will increase.

3.3. Run-time systems

The problems mentioned above can be solved by avoiding static translation. In recent years, several dynamic AOP systems were developed that do not perform static translation. RtJAC [4] performs dynamic translation using the HotSwap capability of JPDA [5] (Java Platform Debugger architecture). HotSwap allows redefinition of classes already loaded into the Java Virtual Machine (JVM). However, redefinition is not allowed when the new definition of the class contains any structural modifications in contrast to the old one. Because of this, RtJAC is not able to perform introductions into classes or changing the inheritance structure.

Two versions of Prose has been released, which are fundamentally different. The initial version of Prose [6] is based on another feature of the JPDA: notification requests can be registered into the JVM for certain events during the program execution. Although this technique does not use static translation, structural modifications still cannot be performed. The second version of Prose [7] uses static translation.

Wool [8] uses a hybrid approach: users can choose between the hotswapping mechanism and the other one used by Prose 1.

Although they are not AOP systems, it is worth mentioning Guaraná and MetaXa. These systems provide metaobject-protocols (MOPs) for Java (see later). Both systems are operating on a modified JVM. Though they can provide maximum flexibility, instrumenting the JVM makes the programs not deployable.

3.4. Summary

In spite of the variety of Java AOP systems, we can state that there is no way to implement such an AOP system in Java that unites the advantages of the individual ones at the same time.

Although the run-time and load-time solutions could be merged with a compile time one to give an easy-to-use and comprehensible interface above them (as done

in HandiWrap [9]), we cannot get rid of the contradiction between the efficiency of dynamic translation and the need for structural modifications. *The implementation technique in use determines the capabilities of the system.*

Another disadvantage of Java is its very limited capability of reflection. Because AOP systems rely on reflection, and because the metaclasses of Java cannot be extended, every AOP system had to elaborate its own reflection system, enforcing the programmer to learn and use the new one.

4. Lisp

In this section I attempt to give an overview about some features of Lisp that support aspect-oriented programming. I also show that CLOS [10] (the Common Lisp Object System) does not suffer from many constraints of the Java language and its virtual machine.

4.1. Extensibility

As John Foderaro[2] said, "Lisp is a programmable programming language." Lisp was meant from the start to be an extensible language. It has a very limited grammar providing hardly more than parantheses and some constants. The language itself consists mostly of a set of functions, which are just as user-defined ones. Furthermore, Lisp functions are represented by Lisp data structures, allowing to write functions which generate Lisp code. This results in blurred borders between the language and the application. Lisp suggests bottom-up programming [12], "Instead of just writing your program in Lisp, you can write your own language *on Lisp*, and write your language in that."

4.2. Metaclasses

The concept of metaobject protocols (MOPs) stams from CLOS[11]. They are interfaces to the language that make it possible for the users to incrementally modify the language's behaviour and implementation. Although the CLOS MOP was the first MOP implementation, it is still one of the most complete ones. In CLOS every system construct is represented by a metaobject. Metaobjects are objects of metaclasses. The partial hierarchy of metaclasses is illustrated in Table 1. In contrast to Java, which also defines metaclasses (`java.lang.Class` and the classes of the `java.lang.reflect` package), the set of metaclasses in CLOS is not fixed: you can create additional metaclasses by creating subclasses of them. Extending metaclasses fits well into the approach of CLOS, and allows further extensibility of the language.

Thanks to the openness of metaclasses, the new programming constructs of AOP can easily fit into the system. Table 1 illustrates one possible inheritance

[2] John Foderaro is a founder of and Senior Scientist at Franz Inc., a leading supplier of Common Lisp program development systems.

Table 1: Extract from the inheritance structure of metaobject classes

Metaobject Class	Direct Superclasses
standard-object	(t)
funcallable-standard-object	(standard-object function)
metaobject	(standard-object)
generic-function	(metaobject funcallable-standard-object)
standard-generic-function	(generic-function)
method	(metaobject)
standard-method	(method)
slot-definition	(metaobject)
specializer	(metaobject)
eql-specializer	(specializer)
class	(specializer)
built-in-class	(class)
standard-class	(class)
funcallable-standard-class	(class)

Table 2: A possible inheritance structure of AOP metaclasses

Metaobject Class	Direct Superclasses
standard-aspect	(standard-class)
standard-pointcut	(metaobject)
standard-aspectelement	(metaobject)
standard-advice	(funcallable-standard-class standard-aspectelement)
standard-introduction	(standard-aspectelement)

structure of AOP metaclasses. In Fig. 1 the definition of the `standard-aspect` metaclass can be seen.

Using user-defined metaclasses would be very unconvinient if their instances (metaobjects) could only be created using explicit instantiation. CLOS provides a standard way for defining new kinds of metaobjects: the metaclass of instances of a class can be specified as an option to the `defclass` macro. This is illustrated on Fig. 2.

4.3. Introductions

As methods are not encapsulated into classes in CLOS, introduction of methods requires no change on the schema of classes. Introducing fields (slots in CLOS terminology) and changing the inheritance structure, however, requires it. To accomplish it, we can use two standard features of CLOS: reinitializing instances and redefinition of classes. In Fig. 3 you can see a function that adds a new slot into a

```
(defclass standard-aspect (standard-class aspect)
  ((pointcuts :reader get-pointcuts :initarg :pointcuts)
   (weaving-rules :reader get-weaving-rules :initarg :weaving-rules)))
```

Figure 1: The standard-aspect metaclass

```
(defclass aspect-foo () ()
    (:metaclass standard-aspect)
    (:default-initargs :pointcuts here-comes-the-pointcut-definition))
```

Figure 2: The standard way of instantiating metaclasses

class.

Redefinition of a class happens immediately, and it affects not only the newly created instances, but the existing ones as well. The function in Fig. 3 adds a new slot into the `c` objects by reinitializing their metaclass. As it can be seen, the arguments passed at redefinition are the original attributes of the class except the direct slot list which is extended with the new slot now. The inheritance structure can be modified in the same way.

4.4. Weaving advices

Similarly to introductions, weaving advices can also be done by reinitializing metaobjects. In contrast to Java, in CLOS the code of methods (lambda expressions) is accessible from the program itself, giving the user the ability of its manipulation. As lambda expressions are represented as lists, it is easy to examine or modify them. Similarly to introductions, weaving means reinitialization of method metaobjects.

4.5. Usage

In the previous part of this section I showed that new language constructs of AOP can be added to CLOS in the same way as CLOS added the object-oriented extension to Common Lisp. Now, I will show that defining aspects can be carried

```
(defun add-field-f (c name typename initializator)
  (reinitialize-instance c
     :name (class-name c)
     :direct-superclasses (class-direct-superclasses c)
     :direct-default-initargs (class-default-initargs c)
     :direct-slots (cons `(:name ,name :type ,typename :initform ,initializator)
                         (mapcar #'create-slot-def (class-direct-slots c)))
     :documentation (documentation c)))
```

Figure 3: Introduction of a new slot

```
(defmacro defaspect (name superclasses slot-defs pointcut-defs &rest rest)
  (let* ((metaclass (find-if #'(lambda (x) (eq (first x) :metaclass)) (second x)))
         (rest (if metaclass rest `(:metaclass ,metaclass ,@rest))))
    `(defclass name superclasses
       slot-defs
       (:pointcuts (dolist x ,pointcut-defs (defpointcut ,asp ,@x)))
       ,@rest)))
```

Figure 4: The defaspect macro

```
(defaspect aspect-1 ()
  ;; just like by classes, slot definitions could be written here

  ;; advice declarations:
  (:after (or (call "move-by" figure-element integer integer)
              (call "set-p1" line point) (call "set-p2" line point)
              (call "set-x" point point) (call "set-y" point point))
     (display-update))

  (:around (and (execution "pay" employee) (args a))
     (print "before paying")
     (setq a (* a 1.5))
     (let ((ret (proceed)))
        (print "after paying")
        ret))

  ;; the system can be extended further:
;  (:metaclass standard-aspect)
)
```

Figure 5: Defining aspect using the defaspect macro

out in a very natural way, as easily as with AspectJ. Fig. 4 illustrates the `defaspect` macro, using which resembles the use of `defclass`, but also allows advice definitions in a similar way to AspectJ. Fig. 5 shows a sample use of `defaspect`. Note that - in contrast with Java AOP systems - arguments of advices are passed by reference. Although CLOS does not support this passing mode, utilizing the power of macros and dynamic scoping, it can be implemented in a simple way. The implementation is not dicussed here.

5. Conclusion

As extensibility is one of the main characteristics of every Lisp system, implementing an AO framework in Lisp itself is self-evident. The need of the aspects to access or change some values in the base program is a typical issue of dynamic scoping, which is a basic feature of Lisp systems. Through the reflective capabilities of Lisp and the CLOS Metaobject Protocol, the structural and behavioral changes described by the aspects can be done very simply, within the language itself. No

special compiler, virtual machine or low-level manipulation of the program is required. Additionally, the system is compatible with the existing programs, and it is working at run-time, providing the highest flexibility that is possible. Additionally, the system can be a base of further extension.

The paper showed that CLOS can be easily extended to provide a flexible and easy-to-use AOP system, which do not suffer from the limitations of similar systems based on Java. Thanks to the extensibility and dynamics of Common Lisp, the implementation is clear and simple.

References

[1] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C. , Lopes, C., Loingtier, J. Irwin., J. *Aspect-Oriented Programming.* In 1997, European Conf. on Object-Oriented Programming (ECOOP, '97), Springer Verlag, 1997, 220-242.

[2] Xerox Corporation: *The AspectJ Programming Guide.* http://www.eclipse.org/aspectj

[3] Pawlak, R., Duchien, L., Florin, G., Legond-Aubry, F., Seinturier, L., Martelli, L.: *Jac: An Aspect-Based Dynamic Distributed Framework*, 2002, http://jac.aopsys.com/doc/JAC.pdf

[4] Espák, M.: *Improving Efficiency by Weaving at Run-time* 5th GPCE Young Researchers Workshop, 2003, Erfurt (Intl. Conf. on Generative Programming and Component Engineering)

[5] Sun Microsystems: Java Platform Debugger Architecture http://java.sun.com/j2se/1.4.1/docs/guide/jpda/

[6] Popovici, A., Gross, T., Alonso, G.: *Dynamic Weaving for Aspect Oriented Programming* In: 1st Intl. Conf. on Aspect-Oriented Software Development, Enschede, The Netherlands, Apr. 2002.

[7] Popovici, A., Alonso, G., Gross, T.: *Just-In-Time Aspects: Efficient Dynamic Weaving for Java.* In: 2nd Intl. Conf. on Aspect-Oriented Software Development, 100-109. Boston, USA, March 2003.

[8] Sato, Y., Chiba, S., Tatsubori, M.: A selective, just-in-time aspect weaver in *Proceedings of the 2nd Intl. Conf. on Generative Programming and Component Engineering.* Erfurt, Germany. Springer Verlag New York, Inc. 189-208.

[9] J. Baker and W. Hsieh. Runtime Aspect Weaving Through Metaprogramming. In 1st Intl. Conf. on Aspect-Oriented Software Development, Enschede, The Netherlands, pages 86-95, Apr. 2002.

[10] Steele, G. L. Jr.: *Common Lisp The Language 2nd Edition* Digital Press, Second Edition, 1984.

[11] Kiczales, G., de Rivieres J., Bobrow G.: *The Art of the Metaobject Protocol*, MIT Press, 1991.

[12] Graham, P.: *On Lisp*, Prentice Hall, 1993

Postal address

Miklós Espák
Department of Information Technology
University of Debrecen
H-4010 Debrecen
P.O.Box 12. Hungary