The Watson Subject Compiler & AspectJ (A Critique of Practical Objects)

Mark Skipper (mcs@bcs.org.uk). Department of Computing
Imperial College of Science Technology and Medicine,
180 Queens Gate, London SW7 2BZ, UK.

1 September 1999

Abstract

Programming tools now exist that support both Subject-oriented and Aspect-oriented programming. Some simple experiments with these tools have revealed insights into the similarities and differences between SOP and AOP in terms of their assumptions about the software development process.

1 Introduction

Subject Oriented Programming (SOP) [7] and Aspect Oriented Programming (AOP) [10] are two approaches to solving problems in software development relating to the difficulty in cleanly dividing a system into modules to achieve separation of concerns [15].

The problems have to do with the fact that conventional approaches provide structures for organising software that are different from the structures developers find most comfortable when organising the concepts they perceive in the problem domain. The problems are particularly difficult because these structures are required to serve both as a system of conceptual classification to aid understanding (as taxonomy does in biology) and as a structural framework to support fabrication (as scaffolding does in building). An experienced developer uses intuition to predict what constructs will arise in a development project but, even with intuition, refactoring is a regular and vital activity in the development process [4]

SOP supports separation of concerns during the development of a system by allowing the system to be decomposed into a number of component systems which are combined according to a set of rules to give the resultant system [7]. AOP supports separation of concerns during the development of a system by allowing some of the aspects of the system to be separated out for separate development and then combined to give the resultant system [11]. The Watson Subject Compiler (WSC) [6, 9] and ASPECTJ [1, 2] are prototype tools based on SOP and AOP. Experience with these tools gives some idea of the benefits that might be achievable by adopting these approaches even if the tools themselves do not support all features of their respective techniques.

In SOP a system under development comprises a number of modules (subjects) and a set of rules that describes how to put the modules together. The subjects are created equal. Nothing in the SOP approach itself separates any module from the others or requires that it be given special attention. In AOP a system under development comprises the base code module and a number of aspect modules. The aspects contain code fragments known as *cross-cut actions* that encode the aspect's concern with respect to the base module. They also contain the specification of *cross-cuts* on the base module that say where and how these actions should be applied. This is illustrated in Figure 1. In the figure keys represent information about which constructs correspond and wrenches represent information about how corresponding constructs should be combined. In SOP the the keys and wrenches are gathered together, outside the subjects, to form the composition expression. In AOP the keys and wrenches are inside the aspects themselves.

Experiments with these tools reveal insights into practical development issues. In this position paper I will briefly describe some experiments to compare the SOP and AOP implementations and conclude with a discussion of methodological issues that arise.

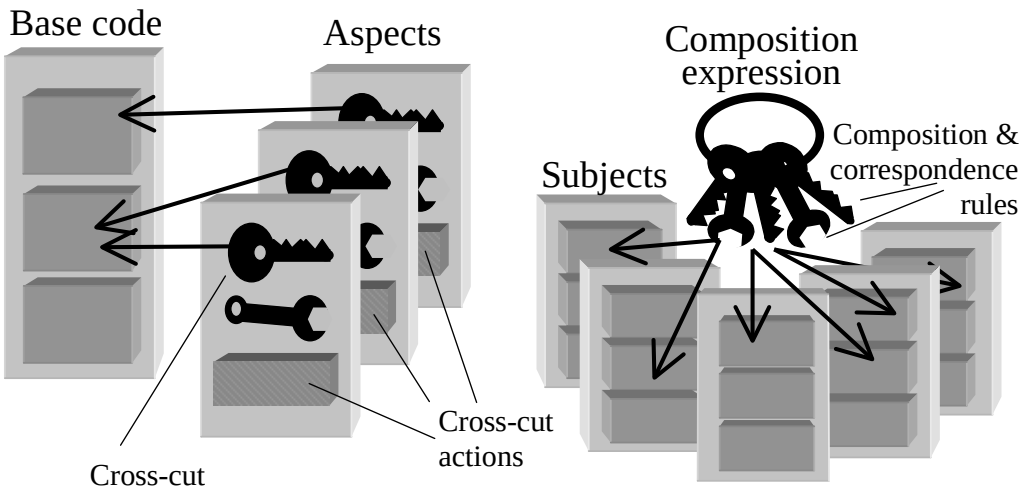

Figure 1: Aspects and Subjects in development.

2 Comparing SOP and AOP

In order to test the expressiveness of the SOP and AOP implementations each was used to encode some simple paradigmatic examples of the other. The results were obtained by examining the obstacles to such encoding rather than, for example, by comparing the execution traces of the resultant programs. This makes those results inherently subjective: what seems difficult to me might seem trivial and obvious to another developer or, perhaps more likely, to members of the development teams of the tools used. Nevertheless I present those experiment here in the belief that they bring to light some interesting insights into the philosophies of SOP and AOP.

The experiments do not deal with the dynamic features of ASPECTJ since WSC does not include facilities for dynamic composition. It seems likely that run-time mechanisms similar to those used by ASPECTJ to register objects with their aspects could be encoded as a subject and composed into a system. Though such a comparison would not be a 'fair contest' in the sense that would not be comparing like with like, it might reveal interesting features of the approaches and so is not ruled out as a future experiment.

2.1 Materials and methods

A series of tiny example programs were written using WSC and ASPECTJ in order to determine whether it is possible to encode congruent structures of composition using the two tools. One set of programs focused on using WSC to achieve the equivalent of before- and after-actions in ASPECTJ. The other set focused on using ASPECTJ to support a subject-oriented style of modularity where all component modules are equivalent, *i.e.* without the base-aspect split.

The SOP tool used was VisualAge for C++ version 4.0 with extensions to support SOP. The AOP tool used was AspectJ version 0.3beta. Since the SOP tool is based on C++ and the AOP tool on Java the example programs were encoded using very basic OO features to avoid artifacts arising from differences between these languages. The constructs used were Classes with non-static integer member variables (attributes), virtual non-static member functions (methods) with integer parameter and return values (or `void` where appropriate). Output was achieved using `cout` (C++) and `System.out` (Java). No exception handling was included.

Wherever possible the same names were used for classes, methods, etc. When encoding AOP structures as subjects the input subjects were named `aspect` and `base`. The output subject was named `comp`.

In ASPECTJ, a static cross-cut action can be prepended or appended to a base method `foo` in class `C` by specifying a advice action with one of the `before` or `after` modifiers and with the `foo` method of class `C` as the target.

In SOP the cross-cut action becomes just another method (`fooB`) in a different subject. The compo-

sition expression must specify that `fooB` should be executed before `foo` when those methods are composed. This is achieved by including an appropriate `ConstrainOrdering` clause which imposes ordering constraints on the component method bodies of a composed method.

When a composite method is formed by composing a number of component methods it may be required that their return results are collected and passed to a function whose job it is to reduce the collection to a single value. This is done by specifying a reduction function in the composition expression. A reduction function is a user-defined function that accepts an array whose elements are of template type and returns a value of the same type. A reduction function rule associates a composite method with a named reduction function. At run-time this function is used to generate a single return value from the values returned by the component methods bodies.

More details of these experiments can be found on the web[1].

2.2 Findings

ASPECTJ allows cross-cut actions to be prepended or appended to base methods. The code of prepended actions can access and modify the actual parameters of the base method's invocation in such a way that those modifications are visible to the base method code when it is executed. The code of appended actions can access and modify the actual parameters in such a way as to change the result value returned by the base invocation. In WSC a method playing the role of a cross-cut action can be prepended or appended to methods playing the role of base. The code of a prepended method can access and modify the actual parameters of its invocation but changes are not visible to method bodies that run after it. The code of an appended method can access the actual parameters and specify a return result but it cannot access the result of any of the method bodies that have executed before it. It is possible to use the reduction function facility to insert code which has all the return values available and thus to arrange for the result of one of the appended method bodies override that returned by the 'base' method. Such reduction functions, however, may not have direct access to the parameter values.

An AOP system normally comprises base code module and a number of aspects. But it is possible, with certain limitations, to move even that base code out into an aspect too. The base code is made up of classes containing attributes and methods which could just as well have been defined with `introduce` actions from one or more aspects. This leaves just the empty base classes to form the base. A construct in a subject may correspond to other constructs in zero or more of the other subjects involved in the composition. Corresponding constructs must be combined which means there must be way to specify their composition. SOP allows a number of composition mechanisms to be used including for example (for methods): override (one body is selected for use and the others discarded), non-deterministic composition (all bodies are to be executed in some order) and sequential composition (all bodies are to be executed in a specific order). In ASPECTJ [2] two such mechanisms are available to control the interaction between aspects: If two aspects introduce methods with the same name to the same base class one of them, selected nondeterministically, will override the other[3]. The `before` and `after` modifiers allow, with some limitations, specification of sequential composition. If two method bodies defined in different aspects are to be combined in a particular order they may be encoded as cross-cut actions for some base method. The one that should come first must have the `before` modifier and the one that should come last must have the `after` modifier. The target method must, however, exist in the base and should be empty. This can be handled using the odd nondeterministic override behaviour described above. Both aspects can, in a symmetrical way, ensure that the existence of the empty base method they require by introducing it. Since both aspects try to introduced empty methods it does not matter that one overrides the other. This technique, however, does not scale; if more than two aspects define code fragments that correspond to the same base method their ordering cannot be controlled: the only available modifiers are `before` and `after` and there is no way to specify the order of actions with the same modifier[4].

[1] `http://www.doc.ic.ac.uk/~mcs98/research/aopsop/`

[2] Version 0.3

[3] This rather otiose behaviour was a feature of early versions of the tool. Since version 0.4 this situation is treated as an error unless one cross-cut explicitly dominates the other in which case the dominant action will override [2, 8].

[4] Later versions of ASPECTJ include the concept of cross-cut domination which, amongst other things, gives the developer control over the order of actions with the same modifier.

2.3 Discussion

SOP and AOP are very similar in terms of their core technologies: An OO system is divided into modules that cut across the dimensions of classification inherent in conventional object-oriented programming. The specification of a system includes details of what code fragments (classes, variables, method bodies, etc.) correspond and how they should be put together. The modules can the be linked by an automated tool (compositor/weaver). These similarities are not surprising since the approaches share a motivation to improve software modularity. The differences between SOP and AOP are in the details of their realisation. These arise from different assumptions, perhaps implicit assumptions, about how software developers will work with such improved facilities for modularity.

The modularity mechanism provided by WSC can be used to simulate a general AOP style of development with a few restrictions. Since this experiment used toy examples it has not been possible to tell if these restrictions would be a serious obstacle in practise.

Fundamental to AOP is the existence of the *base*. The base forms a framework onto which the aspects can hang their cross-cutting actions as one might hang strings of lamps on a tree at Christmas time. Writing SOP style in AOP requires at least the empty base classes whose names can be given to specify join points. As John Lamping points out [12] the main role of the base is to provide the vocabulary that aspect developers use to identify the points where their aspects concerns apply in the system under development. This vocabulary must exist (though it need not be implemented as code or even design) before the system's aspects can be conceived and designed. John says that this vocabulary is formed of the operations (methods, procedures, functions, etc) that correspond to base concepts. Operations provide a convenient granularity for the base vocabulary but they are not the only choice. A more fine-grain base results from using statements or expressions, this has been discussed elsewhere [14]. A more coarse grain vocabulary comes of using the classes as was the case in this experiment. From the point of view of the semantics of the system, the names of these classes are irrelevant. But in AOP development these names are important as they form the vocabulary with which the developer specifies the interaction between aspects. In an attempt to eradicate the base we might conceive a modified weaver that includes a source of new empty classes with which to weave the aspects it processes. Each time a cross-cut names a new class as its target, the appropriate empty class would be generated and the advice woven into it. But even with such an enhanced weaver it would not be possible to program in the SOP style without a common vocabulary established in some other way. In order that two constructs in different aspects be combined they must correspond which means they must name, as their targets, the same base construct[5]. In order for this to work with out enhanced weaver the developers of each aspect need to know the construct naming vocabulary used in the other aspects with which it will be woven. It would be burdensome to examine the text of all other aspects in order to determine this. In SOP this vocabulary information is made available in the form of a kind of public interface called the *subject label*. Subject labels are generated automatically for each module and include information about only the names and structure of composable classes and their members[13].

We may suppose that, in an ideal situation, when doing AOP with ASPECTJ, aspect developers can work separately needing to know only about the base concepts and the aspect under construction; that all interesting cross-cuts are between an aspect and the base, rather than between one aspect and another. In a development with base and n aspects, there would be n interactions between aspect and base. In practise, however, there will be inter-aspect interactions; Recent developments ASPECTJ, such as cross-cut domination, reflect this reality.

The base provides a common vocabulary to all aspects and thus it is a global objective frame of reference for the system development. In AOP this base vocabulary is reified as the base code and contained within one module of the system. In pure AOP any complexity within the base code must be dealt with using the structuring mechanisms of the prevailing paradigm: aspects can't help since if a concept were included in an aspect it would no longer be base. Such purism is not practical, however, as aspects can, in general, extend the base. Consider, for example, using an aspect to add a cache to a web browsing application. After the aspect has been applied the concept of the cache is part of the common vocabulary of the system and should certainly be treated as base by any other aspects that have to do with, for example, file handling or security. This dichotomy comes to light when we consider development as a serial pro-

[5]The name of a member includes the identifier of the member itself and that of its class.

cess where aspects are applied in some order.

Subject-oriented decomposition is not constrained by the need to identify a base module: all modules are potentially equal. The set of modules is determined only by the set of perspectives or concerns that are considered to be important in the system. The cost of this is that any module may interact with any other. In a system with n modules there are $\frac{n^2-n}{2}$ potential inter-module interactions. The SOP view is that there is no global objective frame of reference and that all code is conceived, designed and developed subjectively from a particular perspective and therefore with its own vocabulary. Unlike in AOP the correspondences cannot be specified within the modules as each module knows only its own vocabulary and not that of the modules with which it will be composed. Composition rules are, therefore, separate from the modules, in a global name-space where they can translate between the subjective vocabularies of the component modules in order to specify their join points.

Writing composition rules is, in general, a very difficult task. The rule writer must have a global view and understand all perspectives and their associated subjective vocabularies. Where in AOP correspondence is established by referring to the same name in the shared vocabulary in SOP where two corresponding constructs might be named differently correspondence must be decided by some semantic knowledge. In practise, of course, much of this complexity is avoided. It is frequently the case that a module is created as an extension or augmentation of one or more existing modules. In these cases the extending module shares the vocabulary of the extended modules which simplifies the task of the rule writer. If care is taken in the construction of such an extending module a stock rule can be used such as "name matching" which establishes correspondence based on common names. In this case the extended modules play the role of base and the extending module of aspect. The complexity can be further reduced by composing the extending and extended modules as a sub-unit of the system. The result is a new component module which can be composed with others by a simpler rule than would be needed if all modules were composed in a single step. If each module in the n module SOP system were composed with exactly one other at a time there would be $n-1$ composition steps rather than 1 but similarly there would be only $n-1$ interactions between modules for the rule writer to consider. In general we must assume the number of interactions to be somewhere between $n-1$ and $\frac{n^2-n}{2}$.

3 Conclusion

SOP and its associated tool give the developer much freedom to combine the results of differently conceived and executed software development work. The cost of this freedom is the danger of complex composition rules to combine modules rendered incompatible by the lack of a global common vocabulary to define their join points. Concern-based decomposition can lead to modules with subtle dependencies amongst them[3]. These dependencies impose sequencing constraints that the development process must respect, for example, some concerns only exist if other concerns are already present. Recognising and working with these inter-module relationships in a disciplined way should make SOP more effective and simplify the task of writing, or selecting, composition rules. One candidate for such a discipline is the base-aspect division of AOP which might be employed in SOP as an example of a pattern (in the sense of [5]) which prescribes a particular composition rule (name matching) when a particular topology (some modules encoding base concepts and others aspects that cross-cut them) is present in the modules. It would be damaging, however, to require that the base concepts be confined to a single module. The base concepts might be spread over a number of modules or the pattern could occur many times in a system with different modules playing the base and aspect roles in each.

Dependency between modules is fundamental in AOP: all aspect modules depend on the base module. The base-aspect split is a discipline with the potential to simplify development of systems with concern-based modularity. It reifies the global common vocabulary into the code of the base module. ASPECTJ makes it easy to link all modules in one step. This is perfect for the kind of pure AOP where each aspect module knows only about itself and the base. In practise, however, aspects may be used to encode a series of incremental changes where the early ones redefine the base vocabulary for the later ones. In this case (later) aspects will cross-cut concepts that are encoded in other (earlier) aspects and therefore it must be possible to specify join points and cross-cuts between concepts defined only in aspects. Recent developments in ASPECTJ, such as cross-cut domination, go some way to address this requirement. But, while the new mechanisms that handle inter-aspect relationships are different from those that

handle the aspect-base relationship, the problems they address are not orthogonal. Aspect-oriented developers would do well to be aware that base concepts and the associated vocabulary that they define are central to AOP, but that they should expect to find these concepts and terms in the aspects as well as in the base.

The experiments performed in this study use only toy programs and cannot not give insight into the issues that might arise when SOP and AOP tools are deployed in real development projects. In the end it is that kind of experience that will reveal the strengths and weaknesses of the approaches and of the tools that support them.

References

[1] The AspectJ primer: A practical guide for programmers. `http://www.parc.xerox.com/aop/aspectj/primer`, May 1999. Draft.

[2] AspectJ lannguage specification. `http://aspectj.org`, 1999. Modified 10 August 1999.

[3] L. Carver. Personal communication, Aug. 1999.

[4] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Object Technology Series. Addison Wesley, 1999.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing. Addison-Wesley, 1994.

[6] W. Harrison, M. Kaplan, A. Katz, V. Kruskal, E. Lan, and H. Ossher. Prototype support for subject-oriented programming. Position Paper for OOPSLA '94 Subjectivity Workshop, 1994.

[7] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of OOPSLA '93*, pages 411–428. ACM, 1993.

[8] E. Hilsdale. Cross-cut action domination. Private Communication, Aug. 1999.

[9] M. Kaplan, H. Ossher, W. Harrison, and V. Kruskal. Subject-oriented design and the Watson Subject compiler. Position Paper for OOPSLA '96 Subjectivity Workshop, 1996.

[10] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. Videria Lopes, C. Maeda, and A. Mendhekar. Aspect-Oriented Programming - a position paper from the Xerox PARC Aspect-Oriented Programming project. `http://www.parc.xerox.com/spl/projects/aop/`, 1997.

[11] G. Kiczales, J. Lamping, and A. Mendhekar. Aspect-oriented programming. In *European Conference on Object-Oriented Programming ECOOP*. Springer-Verlag, June 1997. LNCS 1241.

[12] J. Lamping. The role of the base in aspect oriented programming. `http://wwwtrese.cs.utwente.nl/aop-ecoop99/`, 1999. Position paper for the ECOOP'99 workshop on Aspect oriented programming.

[13] H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal. Subject-oriented composition rules. In *Proceedings of OOPSLA '95*. ACM, 1995.

[14] H. Ossher and P. Tarr. Operation-level composition: A case in (join) point. Position paper for ECOOP'98 Aspect-Oriented Programming Workshop, 1998. `http://wwwtrese.cs.utwente.nl/aop-ecoop98/`.

[15] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, Dec. 1987.