On Aspect-Oriented Approaches

Mattia Monga

Università degli Studi di Milano
Dip. Informatica e Comunicazione
Via Comelico 39/41, I-20135 Milano
mattia.monga@unimi.it

Abstract. Several approaches to program construction call themselves aspect-oriented. However, there is still no common agreement on what should be considered aspect orientation. The appealing proposal by Filman who summarized it in *quantification* and *obliviousness* is considered simplistic by who views aspect orientation as a true new paradigm of software development. This paper drafts a model of aspect-oriented approaches, by showing how major examples of aspect orientation fit in it. The proposed model claims that any aspect-oriented approach is essentially a two-step process: a system is initially designed in a traditional way, while a second step is introduced in order to predicate about the entities defined, explicitly or implicitly, during the first step. Therefore, aspect-oriented approaches are in principle devoted to software evolution or augmentation, in contrast to reflective approaches, which uses models of the system during system computations themselves.

1 Introduction

Separation of concerns is a very general and very powerful principle that applies to any large and complex human activity. It is especially used in software to express the ability to identify, describe, and handle important and critical facets of a software system separately. Concerns are always related to a goal a stake-holder wants to achieve with a software system or to anticipations or expectations he or she has on a system. A concern can be seen as a perspective that is taken by a stake-holder on a system in a given development stage [6].

Well organized software systems are partitioned in modular units each addressing a well defined concern. Such parts are developed in relative isolation and then assembled to produce the whole system. A clean and explicit separation of concerns reduces the complexity of the description of the individual problems, thereby increasing the comprehensibility of the complete system [10].

Traditionally, programming languages provide constructs to partition the software in modular units of functionality. Such parts are then assembled to get the desired functionality of the whole system. Traditional languages provide *procedures* and *functions*. Object-oriented languages break up programs in objects isolated by *class encapsulation*: this boundary limits the influence of pieces of code to localised regions; moreover, *inheritance* allows one to incrementally

evolve components by adding new features or redefining existing features.
However, sometimes a concern is not easily factored out in a programming unit, because it *cross-cuts* the entire system, or parts of it. Synchronization, memory management, network distribution, load balancing, error checking, profiling, security are all *aspects* of computer problems that are unlikely to be separated in well defined units. Common experience suggests that every decomposition of a complex system leaves some concerns *scattered* among components. Scattering is a problem because it hinders the possibility to reason about a concern in isolation, by temporarily ignoring what is currently irrelevant. Thus, the problem is clear: which means are needed to clearly separate every relevant concern? *Aspect* is the name of a concern that, in a given context, is not easy to isolate from the others and *aspect-oriented approaches* are those that propose a technique able to achieve a higher degree of separation of concerns.
I am not happy with the above definition. On the one hand it obfuscates the distinction between the problem and its tentative solution. Obviously enough, separation of concerns is not a new problem. It has its roots in the very essence of the western thought: Aristotle used taxonomies as the fundamental tool of his philosophy and Descartes identified *analysis* as one of the four driving principles of his research method. As far as software engineering is concerned Parnas' reflections on program decomposition and information hiding date to the '70s and they are considered at the core of the discipline. On the other hand, if aspect orientation is something able to separation difficult concerns, it is clearly a good thing and we all should do it, since an aspect-oriented system is, by definition, a system in which every relevant concern is separated enough to be managed in isolation. I believe that such systems do not exist today, although several systems claim to be aspect-oriented. We should sharply distinguish between the problem, *separation of concerns,* and the techniques we use to try to cope with it. In this paper I claim that some of them share some common principles and they deserve a common name, namely *aspect orientation.* This should be useful in understanding when an aspect-oriented approach is appropriated and when some other approaches should be chosen to solve the problem at hands. Pragmatically, aspect-oriented approaches are powerful tools that software engineers should have in their portfolio, but only if we will able to define their essence, their power, and their weaknesses, we will be free from the recurrent search for the silver bullet.
The paper is organized as follows: Section 2 presents what I consider to be the essential part of aspect-oriented approaches, providing some examples of a few popular aspect-oriented systems described under this perspective and Section 3 draws some conclusions.

2 A model of aspect orientation

2.1 Aspect orientation à la AspectJ

The notion of aspect-oriented programming was introduced by Kiczales et al. in [8]. Their approach was successfully implemented in AspectJ [1] by Xerox

PARC. AspectJ aims at managing tangled concerns at the level of Java code. AspectJ allows for definition of first-class entities called `aspects`. This construct is reminiscent of the Java `class`: it is a code unit with a name and its own data members and methods. In addition, aspects may *introduce* an attribute or a method in existing classes and *advise* that some code is to be executed `before` or `after` a specific event occurs during the execution of the whole program. Aspect definitions are *woven* into the traditional object-oriented (Java) code at compile-time. Nevertheless, events that can trigger the execution of aspect-oriented code are run-time events: method calls, exception handling, and other specific points in the control flow of a program.

The AspectJ way to provide support for encapsulating otherwise cross-cutting concerns is based on:

1. a syntactic extension to Java for isolating aspect-oriented code;
2. a language for identifying *join points* where advices should be introduced;
3. a compile time *weaver* responsible to mix aspects with the rest of the code in order to produce the running system.

Integration, or weaving in the aspect-oriented jargon, can be in principle performed in different ways and at different times. In fact, it can be done at link-time [2], at run time [11], or at deployment-time [2].

The nature of the join points strongly affects the properties of the integration: its flexibility, the ability to understand the integrated system in terms of its components, reusability of components, and the nature and complexity of weaver and other supporting tools.

In principle, the various aspects should not interfere with functional code, they should not interfere with one another, and they should not interfere with the features used to define and evolve functionality, such as inheritance. Currently, non interference in presence of class evolution is really hard, and programmers should be very careful in writing aspects that make use of the implementation details of classes as little as possible if they want to be able to reuse their aspects. Moreover, it is still not clear how to cope with the difficult problem of aspect interaction (see [4, 7, 3] for some work in progress and discussion). An interesting point to be noted is that, while aspects say something about classes, classes are unaware of aspects, i.e., it is not possible to name an `aspect` inside a `class`: aspects are conceptually *a posteriori* with respect to classes, they augment the system by some functionality. Although advices may affect aspect code, advices assume the existence of some working code to be "advised", i.e., convinced to do something else with respect a prior idea.

2.2 The core of aspect orientation

In a widely cited paper [5], Robert Filman compares aspect-oriented programming to the event driven programming paradigm. He claims that aspect orientation is fundamentally different from publish-subscribe because it is characterized by both

1. *quantification:* the ability to do something when a specific property occurs to be true;
2. *obliviousness:* programmers are more or less unaware of what can be triggered by their code;

On the contrary, in event driven systems, programmers expose explicitly which properties can be quantified and other components explicitly subscribe themselves to be notified when the property becomes true. While generally considered insightful, Filman's work is sometimes depicted as simplistic [12] since it neglected the importance of the join-point model (i.e., what can be quantified and what can be exposed by quantification itself). Moreover, the very concept of obliviousness is quite controversial among aspect orientation experts. A loosely regulated obliviousness as the one allowed by AspectJ constructs may cause turbulent *ripple effects* that in general force programmers to take into account most of the statements of a system in order to understand the properties of the system they are building. As a consequence, several scholars are proposing techniques to mitigate, discipline, or even avoid obliviousness in aspect oriented approaches.

Thus, it is still not clear what exactly is at the core of aspect orientation. I suggest that, while quantification is important, the essential part of any aspect-oriented approach is the *two-step process*, by which quantification is exploited. The conceptual model of aspect orientation I have in mind is depicted in Figure 1. The system is described, designed, or implemented in any coherent way. However, some issues are more or less *orthogonal* to this description, design, or implementation. Thus, in order to describe, design, and implement this augmentations, a model (an abstraction) of the system is considered. The original conception of the system and any augmentation are put together by some algorithmic device that, by knowing the ontology of the model, is able to weave augmentations to produce a running system. It is worth noting that, since the model is an abstraction of the system itself it is more general: as a consequence of this generality, an augmentation concerning a general element may also affect some entity that is not part of the original system. For example, if one states that "every function call should be traced by a `printf`", the model of the system is the function call graph, and a universal quantification implies the affection of `printf`s too; in order to avoid infinite recursion, one should state the augmentation as "every function call in the original system should be traced by a `printf`".

In other words, aspect-oriented approaches are evolution techniques in which one defines a computation Δ that transforms a software system S in a new system S'. Δ can be defined by the tuple

$$< M, A, \mu, \omega >$$

where:

- M is a model of the system obtained by applying the abstraction function

$$\mu : S \to M$$

,

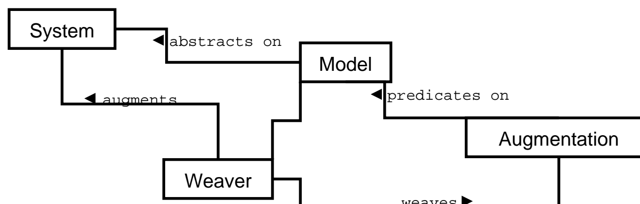

Fig. 1. The essence of aspect oriented approaches

- A is the augmentation of S defined on M
- ω denotes the weaving
$$\omega : S, A \rightarrow S'$$
.

According to this framework, an aspect-oriented approach can be described by stating the features of $< M, A, \mu, \omega >$. Thus, obliviousness about Δ becomes a property of the construction of S, irrelevant for the approach (while still significant when the properties of S and, in general, S' are concerned). Distinct aspect-oriented approaches may differ for the ontology and granularity of M (what can be quantified), for the mapping mechanisms μ (what is considered in mapping: which entities and which perspective, whether a static viewpoint is involved, or the system is considered in its run-time, link-time, deployment-time structure. This is a critical decision that heavily affects information hiding assumptions), for the ontology A (what can be part of the augmentation), and for the weaving mechanism ω.

2.3 Actual aspect-oriented approaches

A couple of representative examples should suffice to give an idea about the framework rationale. For a detailed account of the described approaches, readers are directed to the referenced papers.

AspectJ AspectJ [1] (see also the introductory discussion in section 2) presents a rich M ontology and mapping power. In fact, a wide range of abstraction levels are allowed in what can be considered as a *join point* in a Java system:

- Lexical entities (`withincode`, regular expression on identifiers, etc.)
- Statically known entities (`void func(int)`, etc.) and type relationships
- Run time events (`call`, `execution`, if something happens during a given control flow, etc.)

The mapping μ operates at the level of the Java language specification: for example, it can distinguish between function calls and function dispatching, it can override encapsulation, it takes into account static and dynamic types.

This is the source of several intricacies, since AspectJ programmers have to be aware of both Java and AspectJ subtilties to master their Δ. Augmentations are composed by every legal Java entity and every join point can be augmented before, after, or instead its detection. Weaving ω is performed at compile-time, and cannot be undone.

Hyper/J Hyper/J [13] is a language to define multiple dimensions of concern within a software system written in Java. Hyper/J considers the system as a set of Java declarations (methods, attributes and classes) and provides:

1. a notation to map declaration units to arbitrary *concerns* and concerns to concern dimensions: i.e., each unit is assigned to one or more concern;
2. a notation to describe various correspondences between declarations and definitions;
3. an engine to bind together declarations and definitions according to such correspondences.

Hyper/J considers the decomposition of the system in classes as a dimension of concern (the `ClassFile` dimension), in principle analogous to the others: its creators say that there is no tyranny of the dominant decomposition [13]. The set of all declaration units form a *hyper-space* from which the compiler can cut *hyper-slices* containing all concerns pertaining to a dimension. Hyper-slices can be generated *declaratively complete* including in it all definitions needed for static checking. These slices are eventually integrated in executable hyper-modules by specifying which definitions must be linked to abstract declarations.

Again, essential in the Hyper/J approach is the two step process: programmers write a Java system and then it can be mapped on a new model (the hyper-space). Augmentations are described in term of hyper-slices and the new system is composed by augmented hyper-modules.

3 Conclusions

Nowadays, the main problem aspect-oriented approaches are facing is the need for a better understanding of what properties of S are preserved in the augmented system S'. I believe that in order to solve this hard question, a clearer comprehension in what is essential in an aspect-oriented is key. An important contribution to this goal was Filman's work. However, quantification and obliviousness can be a too restricted, and sometime misleading, view of aspect orientation. On the other hand, other proposals (see for example [9]) describe the richness of the world of aspect-oriented systems, but an abstraction of its essential features is missing. The model proposed above claims that any aspect-oriented approach is essentially a two-step process: a system is initially designed in a traditional way, while a second step is introduced in order to predicate about the entities defined, explicitly or implicitly, during the first step. This is, in a sense, the meaning of cross-cutting, that can be defined only if one assumes an existing structure.

Therefore, aspect-oriented approaches are in principle devoted to software evolution or augmentation. From this perspective, aspect-oriented approaches differ from reflection (although they often leverage on reflective services to implement their abstractions) which, instead, uses a model of the system during the computations of the system itself.

References

1. Aspectj. http://www.aspectj.org.
2. Aspectwerkz. http://www.aspectwerkz.org, 2004.
3. D. Balzarotti and M. Monga. Using program slicing to analyze aspect-oriented composition. In C. Clifton, R. Lämmel, and G. T. Leavens, editors, *Proceedings of Foundations of Aspect-Oriented Languages Workshop at AOSD 2004*, pages 25–29, Lancaster (UK), Mar. 2004. Iowa State University.
4. R. Douence, P. Fradet, and M. Südholt. Composition, reuse, and interaction analisys of stateful aspects. In *Proceedings of the 3rd international conference of aspect-oriented software development*, Lancaster, UK, Mar. 2004. ACM.
5. R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proceedings of OOPSLA 2000 workshop on Advanced Separation of Concerns*, 2000.
6. A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in systems development. *International Journal of Software Engineering and Knowledge Engineering*, 1(2):31–58, 1992.
7. S. Katz. Diagnosis of harmful aspects using regression verification. In G. T. Leavens, R. Lämmel, and C. Clifton, editors, *Foundations of Aspect-Oriented Languages*, Mar. 2004.
8. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997. Springer-Verlag.
9. K. Mehner and A. Rashid. Towards a generic model for aop (gema). Technical Report CSEG/1/03, Lancaster University, Lancaster LA1 4YR, UK, 2003.
10. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.
11. R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. Jac: A flexible solution for aspect-oriented programming in java. In *3rd International conference on Meta-level architectures and separation of concerns*, number 2192 in Lecture Notes in Computer Science, pages 1–25. Springer-Verlag, 2001.
12. A. Rashid. Personal communication.
13. P. Tarr and H. Ossher. *Hyper/J^{TM} User and Installation Manual.* IBM Research, 2000.