A Formal Foundation for Aspect Oriented Software Development

Jorge Fox
Institut für Informatik, Technische Universität München
Boltzmannstr. 3, D-85748 Garching b. München, Germany
fox@in.tum.de

Abstract

Aspects are widely accepted as properties that cross-cut several components in a subsystem. Aspect Oriented Software Development (AOSD) is a challenging field of research. On the one hand, the main problems have been defined and addressed; on the other hand, these problems and their solutions have brought new ones. Since a few years there are mainly two linguistic approaches for aspectual programming languages, namely AspectJ and Compose*. Proposals towards the modeling of aspects, have also been outlined, as well as methods for aspect identification at the requirements stage. Tools for aspect mining in code have also been developed. The efforts of several research groups come together and provide an interesting framework which we believe is aimed at improving current software development approaches and build over the Object Oriented Paradigm. It is nevertheless necessary to precise our understanding of the subject matter of research, the so called aspects, and provide for a clear definition, a syntax and semantics. This paper proposes a basic still precise definition of aspect and outlines an approach to provide aspects with a syntactical and semantic corpus.

1 Introduction

For many serious computer scientists the name AOSD or Aspect Oriented Programming (AOP) is still obscure and the concept of *aspect* ambiguous, to say the least. A number of problems in software engineering have been correctly addressed by Object Oriented Programming (OOP). Basically, OOP manages to handle complexity by providing a useful and very clear concept for programming. Together with it a series of methodologies for software engineering have been developed. It is out of the scope of this work to praise the OOP, it is also needless, but to draw attention to its enhancement.

© A. Gelbukh, C. Yáñez Márquez, O. Camacho Nieto (Eds.)
Memoria del XIV Congreso Internacional de Computación CIC 2005.
ISBN 970-36-0266-5. CIC, IPN, México, 2005, p. 434-444.

The author believes that the enhancement of the OOP lies in comprehending its inherent limitations. The class abstraction, understood as a a collection of types of encapsulated instance variables and types of methods, helps encapsulating closely related behavior. Nevertheless, while doing so, it restricts the possibility of translating a number of concerns, a problem which a program tries to solve (synchronization policies, security, among others), into the class programming abstraction without leaving the concern dispersed in several classes. The cons of Object Oriented Programming (OOP) in relation to composition and concern separation have been identified as in [BAT01, Aks96]. In this work, we would like to add the understanding that such limitations stem not from the technical difficulties related to the OOP, given that it is a rather mature technology, but from the abstraction itself.

Let us explore the limitations of the OOP and the reasons to consider AOP as an extension to it. Object oriented techniques decompose software into modules, in this case classes, because the unit of modularity in OOP languages is the class. Some desired behavior might be common to different classes and therefore ends up spread among several classes. Systemic concerns or concerns that relate to a group of classes, such as security concerns, cannot be encapsulated in a single unit and therefore surface disperse across several classes.

The role of programming languages in shaping the abstractions by which software designers and programmers aprehend and organize software can not be underestimated. This applies for requirements engineering as well. The abstractions that ultimately shape a software are heavily influenced by the underlying implementation paradigm, like the prevalecent class/object concept.

In this evolutionary trend we find more and more conceptual tools, just like objects in OOP provide an abstraction for elements in the real world. Evolving together with the programming languages we find software development methodologies.

Nevertheless, the object abstraction along with the composition mechanisms provided in the OOP entail limitations. These limitations have already been discussed in [Aks96] and more in depth by S. Clarke in [Cla01]. S. Clarke clearly demonstrates that the units of modularisation in the OOP are structurally different from the units of modularisation of requirements specification. This is the source of aspects.

2 A definition of aspect

2.1 Current definitions

It is commonly accepted that *aspects* are "concerns that cut across other concerns". According to the early definitions of aspects, two concerns crosscut if the methods related to such concerns intersect [EFB01]. Therefore being responsible for producing tangled representations that are difficult to understand and maintain [ARA02]. This crosscutting is relative to a particular decomposition [AKLO01]. A concern is any area of interest in a software system. We should

also consider the meaning of the word aspect in order to shape the concept correctly and use the right name. The word aspect has the following meaning:

> *The particular angle from which something is considered. i.e. Perspective*

Moreover, the word aspect comes from the latin *aspectus*, from *aspicere*, that is, to look at. All in all the selected definition and the etymological meaning of the word reflect the idea of a perspective towards an artifact, in this case towards a system. Choosing the word aspect is then consistent with conceiving a system as composed of different parts (subsystems) or angles and allows us to separate such subsystems as areas of interest. In short, to arrange a system in concerns.

A number of difficulties for aspect identification, either at requirements or at other stages of software development stem from a definition of aspect that needs to be made more complete and precise. Let us for instance consider the proposal on early aspect identification as in [AC03]. Their approach towards aspect identification relies on use cases, when a use case extends more than one use case or when a use case is included by one or more viewpoints then it is considered an aspectual use case. There are a number of difficulties associated with aspect identification by doing so, for instance, prioritization of conflicts that stem from different viewpoints is done by hand, and by hand is made also the decision of what is an aspect and what is not an aspect once they identify candidate aspects. It is nevertheless a valuable approach that gives an important insight towards aspect identification.

Moreover, the problem of aspect identification relates to the fact that we need to have an integral view of the problem of concern cross-cutting and consider its context as well. As authors like [Lop04] have already outlined, the problem AOSD solves is one of complexity in today's software applications.

Therefore, in order to provide clarity and later on a new computational abstraction we introduce the following definition which builds on the one presented in [FJ05] and the commonly accepted definition introduced at the beginning of this section.

2.2 Definition

We define an aspect as a requirement that is partially implemented in more than one class as illustrated in figure 1. An aspect is defined as follows:

Definition (Aspect) Given a problem space $\mathcal{P}$
a solution space $SOLSP = \bigcup_{\rho} SolSp(\rho)$
Let $A =_{def} \{r \in \mathbb{RE} \mid \exists\, o_1, o_2 \in \mathbb{CL} : Implements_{\varphi}(r, o_1) \wedge Implements_{\varphi}(r, o_2) \wedge o_1 \neq o_2\}$
where
$Implements_{\varphi}(r, \theta) \Leftrightarrow \forall_{\varphi'} \subseteq \varphi.(\varphi' \models r \Rightarrow \theta \subseteq \varphi')$
$\varphi \models r$ φ *satisfies* r
$\mathbb{RE} = \mathscr{P}(SOLSP)$
$r \subseteq \bigcup_{\rho} SolSp(\rho)$

$\mathbb{CL}_{\varphi}$ is the set of classes in the solution space.

Please note that the link between requirements and aspects has been identified by authors like [Jac03]. B. Tekinerdogan and M. Aksit in [TA01] discuss the concept of solution and problem domain, we refer to them here as solution and problem spaces.

Now consider the following two examples. First, portray an example from the literature. The problem of multiple views on a system is discussed by M. Aksit in [BAT01] as there presented as a characteristic AOP problem that can be solved by composition mechanisms as *Compose** [1].

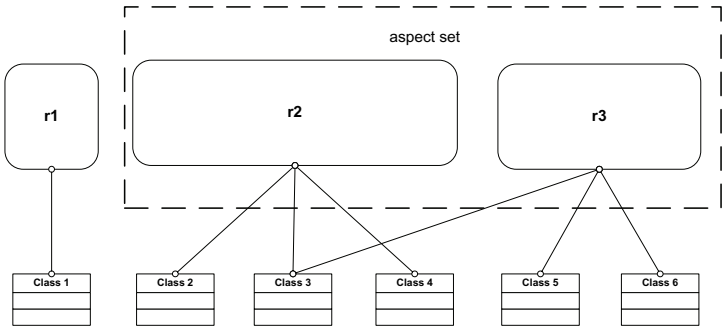

Figure 1: Requirements and aspects in relation to classes

The problem can be illustrated as follows. In [Fox02] the author presents a small E-Learning Support System (ELSS). The system is aimed at facilitating the evaluation of students in a university with online support for learning. Students are supposed to attend a given number of hours of self-learning and evaluate their advance with use of this system. The professors define the tests to be applied by the system and are also interested in having an overview of their students' advancement.

The system, shown in figure 2 is designed for the following users: Professors and Students. We may define two views on the system and based on it restrict access to evaluation-related methods by identifying the user. Restricting access based on the user is a requirement that under OOP is implemented in different classes. We should then define views on the system, the studentView and the profesorView. Table 1 shows the access critical methods that are assigned to each view.

The object abstraction allows us to encapsulate the behavior related to each of the classes. But assume we decide to modify the ELSS and add a new view

[1] http://trese.cs.utwente.nl/

Table 1: Methods per View

studentView	*professorView*
readTest()	enableTest()
answQuestion()	defineAnswer()
checkOwnAnsw()	evaluateStudent()
answTest()	groupAverage()
	eraseTest()

on the system. This view could be the *university exams administration office* and has the duty of enabling a given student to take an examination, as well as a given professor of applying an test. The behavior related to this new view can not be encapsulated in one single new class. That is what we mean by an aspect.

As a second example, consider the same base system from figure 2. In this case, assume the users may access the system remotely. The requirement is that the information related to the exams shall be kept secret to third parties through the communication channel. This can be achieved by adding an encryption protocol to every information send from and to classes `student`, `professor`, and `exam`, the example is being kept simple in order to point at the definition of aspect and the suggested structure in section 4.

Both aspects are finally implemented cross-cutting several modularisation units, in this case, several classes.

Based on this examples, in section 3 we may distinguish two types of aspect, both can be considered by definition 2.2.

3 A Classification of Aspects

The two examples presented above can be considered representative of the typical aspects found in the literature, from synchronization policies, all the way to composition aspects as multiple views, considering as well logging, security, persistence, which are system properties involving more than one functional component and help substantiating our definition. The reader may also refer to [FJ05] for a reasoning on the definitions of aspect, though in that paper we outlined a preliminary definition not considering the problem of multiple views as explored in [BAT01].

We may classify aspects now, based on the composition problems involved in the following two main groups:

1. *Type 1 or Systemic* e.g. multiple views, quality of service. Example 1
2. *Type 2 or Fine Granular (FG)* e.g. security, synchronization, logging. Example 2.

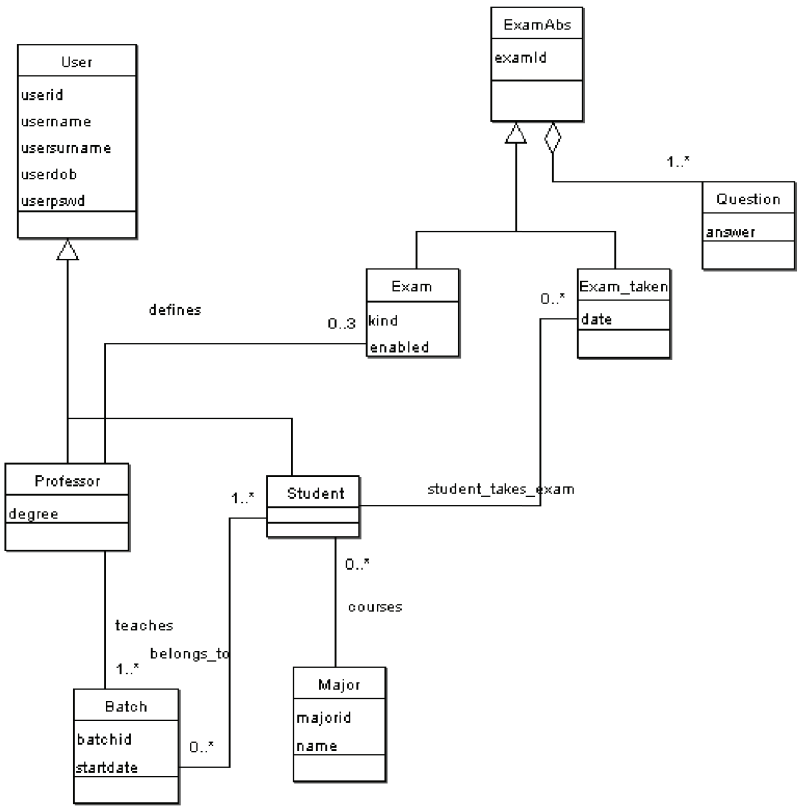

Figure 2: Class Diagram E-Learning System

We call Type 1 Systemic, based on the reason that these aspects influence the decisions shaping or modifying the architecture of a system. While type 2 aspects can be implemented at the level of the methods in the classes. The problems related to the transformation from requirements to a systems design and implementation are outside the scope of this work. However, we must keep in mind that different solutions to the problems posed by aspects, *cross-cutting concerns*, have been achieved. Type 2 aspects may be in some cases solved by using Design Patterns, AspectJ, or Composition Filters, for instance. Type 1 aspects by composition mechanisms as Hyper/J or Composition Filters as already mentioned.

The need for a new abstraction for aspects together with a theoretical framework for the composition problems that these present has not been previously solved, not at least in a comprehensive way as presented here. This paper

therefore proposes the following abstraction for an aspect. We will define in this paper the structure for aspects of type 2. An abstraction for aspects of type 1 will not be defined in this paper.

4 A Structure for Aspects

Just as the Class in OOP allows us to encapsulate data and methods in one unit, we propose to embody the relation between an aspect and the classes affected by it as illustrated in table 2. In order to do so, we need the following elements: an identifier, the set of classes that relate to the aspect, the methods modified by it, and finally a function specifying the aspectual behavior that adds to the selected base methods.

Table 2: The Aspect Abstraction

Aspect
Aspect identifier: Name e.g. Encryption
Set of related Classes e.g. $\mathbb{S} = \{Student, Professor, ExamServer\}$
Set of concerned methods e.g. $\mathbb{BE} = \{Student.answTest, Student.answQuestion, Professor.defineAnswer\}$
Mapping function $f : \mathbb{M} \times \mathbb{M} \rightarrow \mathbb{M}$ $f(n, m) =$ method n modified by method m where m is the specification of the aspect functionality e.g. $f(Student.answTest, sendEncrypt) = Student.answTest \succ sendEncrypt$

The first three elements need no further explanation. We refer to the aspect identifier, the set of classes related to it, as well as the methods that the aspect modifies. In our structure there is neither appending of new data elements to the set of classes, nor the possibility of deviating the flow of the program arbitrarily. Both for preserving the principle of encapsulation and avoiding undesired modifications in a given software. Current approaches as AspectJ allow for arbitrary modification of the flow of the program because of its underlying model for implementing aspects, meaning by this the concept of Join-Point and Advice. For a thorough analysis of the *Join-point+advice* model the reader may refer to [DWL03]. We prefer to avoid the drawbacks of the above mentioned model even at the price of its flexibility and power which, by the way, can be

compared to the use of pointers in the ANSI C Language. In [Pri04] the author explores the limitations of the join-point model as implemented in AspectJ, its drawbacks are mainly the following:

- it may not contribute positively to the modularization of the program,
- it has a negative impact on coupling and readability in a program
- the use of interceptions and introductions changes a program in a non-trivial manner and hinders its unambiguousness

Considering the above our mapping function is conceived in such a way that the unambiguousness (clarity) of a program as well as its modularization are preserved. Therefore, concerning the function mapping the aspectual behavior into the set $\mathbb{BE}$ we have chosen to preserve the original function and modify it as described below.

The function relating the classes given methods and the new method is based on a service architecture approach. On the one hand, we propose to specify the class methods and the behavior of the aspect in a formal language.

4.1 An overview of Focus

This formal language[2] is based on input/output relations on sets of histories of externally observable events. The behavior of a component is described by the relationship between its external input and output histories, defined as streams. It allows us to obtain a *black-box view* of the component in question. It also allows us to distinguish between *elementary* and *composite* specifications. Composite specifications are built from elementary specifications using constructs for composition and network description.

The composition constructs we propose to use are mutual feedback $\otimes$ and piping $\succ$. These are defined as follows:

Given two specifications S_1 and S_2 with disjoint lists of output identifiers:

$os_1 \cap os_2 = \{\}$

We assume that the channel identifiers in

$l = (i_{S_1} \cap o_{S_2}) \cup (i_{S_2} \cap o_{S_2})$

are of the same types L in both specifications. The composition of S_1 and S_2 by mutual feedback $S_1 \otimes S_2$ is defined as follows: $[[S_1 \otimes S_2]] =^{def} \exists\, l \in L^{\infty} : [[S_1]] \wedge [[S_2]]$

moreover $S_1 \succ S_2$ is the special case where $l = o_{S_1} = i_{S_2}$ and $i_{S_1} \cap o_{S_2} = \{\}$

4.2 A Black-Box view of methods and composition

We propose to model the methods in the set $\mathbb{BE}$ (Table 2) as well as the behavior defined in the aspect itself as a black-box with a set of input stream channels and output stream channels as illustrated in Figure 3.

[2]This section is based on [BS01]

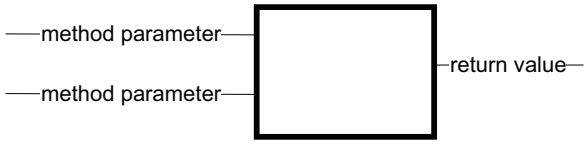

Figure 3: Black-Box View of a Method with two parameters and a return value

The specification of the methods can be made with help of a state machine diagram, as in the example encryption protocol specified by a state machine, provided in [FJ05], or by the specification techniques of the formal language we mentioned above. By reasons of space, this paper focus on the introduction constructs for aspects, and not in the specification of the methods themselves, in order to explain the aspect structure introduced in section 4.

A method is modeled in FOCUS as a component in which every parameter is an input channel, and the return value is an output channel. For reasons of clarity we consider the parameters of a method as *call by reference*.

The composition of the class methods with the aspect is then performed by *mutual feedback* $\otimes$ or *piping* $\succ$. Mutual feedback in case the base method (from set $\mathbb{BE}$) also receives input from the aspect, piping otherwise.

In the example of table 2 we describe the following introduction of aspect *sendEncrypt* in the method *Student.answTest* as the piping of two components as illustrated in Figure 4. The resulting method is defined as:

$[[Student.answTestwEncryption]] = [[Student.answTest]] \succ [[sendEncrypt]]$

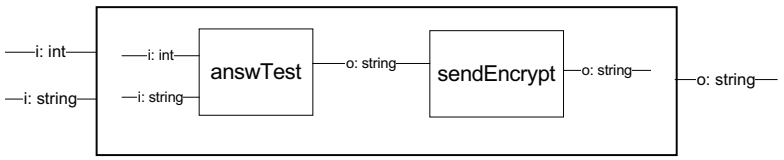

Figure 4: Glass-Box View composition of two components

By modeling the methods of the classes and the aspects we aim at providing them with syntactical and semantic constructs that allow us to further discuss aspect composition and interaction problems. This is outside the scope of this paper.

5 Conclusions

In this paper we have seen that OOP prevents some concerns from being expressed in one single locus, a new abstraction for such concerns is therefore

needed. The aim of this paper was to clearly define aspects, based on the accepted notion. Moreover, our discussion also considers the context in which cross-cutting concerns (aspects) stem, namely from the inherent limitations of the class abstraction.

Equally important, this paper proposes a simple though comprehensive software structure for aspects and outlines two composition constructs for introducing the aspect behavior in the methods of the affected classes. The constructs are safe in the sense of respecting the encapsulation of the base classes and avoiding arbitrarily disrupting the flow of a program. This is an advantage compared to other aspect introduction approaches and further work will be carried out in this line of research.

Finally, a number of challenges in AOSD reside in aspect identification, this research work helps providing a clear definition of aspects and a new abstraction to model them.

Acknowledgements. Special thanks to Jan Jürjens for his help on giving shape to the definition presented here. Thanks to Florian Deißenböck, and Leonid Kof for enriching discussions on the subject.

References

[AC03] Joao Araùjo and Paulo Coutinho. Identifying Aspectual Use Cases Using a Viewpoint-Oriented Requirements Method. In *Early Aspects 2003: Aspect-Oriented Requirements Engineering and Architecture Design, Workshop of the 2nd International Conference on Aspect-Oriented Software Development, Boston, USA, 17th March*, 2003.

[AKLO01] Mehmet Aksit, G. Kiczales, K. Lieberherr, and H. Ossher. Discussing aspects of aspect-oriented programming. *Communications of the ACM*, 44(10):33–38, 2001.

[Aks96] Mehmet Aksit. Composition and separation of concerns in the object-oriented model. *ACM Computing Surveys*, 28A(4), 1996.

[ARA02] Ana Moreira Awais Rashid, Peter Sawyer and Joao Araùjo. Early aspects: A model for aspect-oriented requirements engineering. In *Proceedings of IEEE Joint International Conference on Requirements Engineering (RE 2002), IEEE Computer Society, pp. 199-202*, 2002.

[BAT01] Lodewijk Bergmans, Mehmet Aksit, and Bedir Tekinerdogan. Aspect composition using composition filters. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 357–384. Kluwer Academic Publishers, 2001.

[BS01] M. Broy and K. Stoelen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement.* Springer, 2001.

[Cla01] Siobhan Clarke. *Composition of Object-Oriented Software Design Models*. PhD thesis, Dublin City University, 2001.

[DWL03] Steve Zdancewic David Walker and Jay Ligatti. A Theory of Aspects. In *ACM SIGPLAN International Conference on Functional Programming, Uppsala, Sweden, August*, 2003.

[EFB01] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[FJ05] Jorge Fox and Jan Jürjens. Introducing Security Aspects with Model Transformation. In *Proc. 12th Annual IEEE International Conference on the Engineering of Computer-Based Systems (ECBS 2005), Greenbelt, Washington, 4-5 April*, pages 543 – 549. Computer Society, 2005.

[Fox02] Jorge Fox. E-learning support system. Technical report, National Institute of Small Industry Extension Training (NISIET), Hyderabad, India, 2002.

[Jac03] Ivar Jacobson. Use cases and aspects -working seamlessly together. *Journal of Object Technology*, pages 7–28, 2003.

[Lop04] Cristina Videira Lopes. AOP: A historical perspective. In R. Filman et al., editor, *Aspect-Oriented Software Development*, pages 97–122. Addison-Wesley, 2004.

[Pri04] Franz Prilmeier. AOP un Evolution von Software-Systemen. Master's thesis, Technische Universität München, November 2004. http://home.in.tum.de/ prilmeie/da/.

[TA01] Bedir Tekinerdogan and Mehmet Aksit. Synthesis-based software architecture design. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 143–174. Kluwer Academis Publishers, 2001.