PROCEEDINGS OF THE

ASPECT-ORIENTED PROGRAMMING WORKSHOP

AT

ECOOP'97

Organizers:
Cristina Lopes, Kim Mens, Bedir Tekinerdogan, and Gregor Kiczales

Includes the workshop report and the position papers.

The workshop report was published in *Workshop Reader of the European Conference on Object-Oriented Programming* (ECOOP), Finland. Springer-Verlag LNCS 1357. June 1997.
Its copyright notice follows below:

Aspect-Oriented Programming Workshop Report

Kim Mens[1], Cristina Lopes[2], Bedir Tekinerdogan[3], and Gregor Kiczales[2]

[1] Vrije Universiteit Brussel,
Department of Computer Science, Programming Technology Lab,
Pleinlaan 2, B-1050 Brussel, Belgium
[2] Xerox PARC, Systems and Practices Laboratory,
3333 Coyote Hill Rd, Palo Alto, CA 94304, USA
[3] University of Twente,
Department of Computer Science, Software Engineering,
P.O. Box 217, 7500 AE Enschede, The Netherlands

Abstract. Whereas it is generally acknowledged that code tangling reduces the quality of software and that aspect-oriented programming (AOP) is a means of addressing this problem, there is — as yet — no clear definition or characterisation of AOP. Therefore, the main goal of the ECOOP'97 AOP workshop was to identify the "good questions" for exploring the idea of AOP.

1 Introduction

Mechanisms for defining and composing abstractions are essential elements of programming languages. They allow programs to be composed up from smaller units, and they support design styles that proceed by decomposing a system into smaller and smaller sub-systems.

The abstraction mechanisms of most current programming languages — subroutines, procedures, functions, objects, classes, modules and API's — can all be thought of as fitting into a generalised procedure call model. The design style they support is one of breaking a system down into parameterised components that can be called upon to perform some function.

But many systems have properties that do not necessarily align with the system's functional components. Failure handling, persistence, communication, replication, coordination, memory management, real-time constraints and many others are aspects of a system's behaviour that tend to cut-across groups of functional components. While these aspects can be thought about and analysed relatively separately from the basic functionality, programming them using current component-oriented languages tends to result in these aspects being spread throughout the code. The source code becomes a tangled mess of instructions for different purposes.

This "tangling" phenomenon is at the heart of much needless complexity in existing software systems. It increases the dependencies between the functional

components. It distracts from what the components are supposed to do. It introduces numerous opportunities for programming errors. It makes the functional components less reusable. In short, it makes the source code difficult to develop, understand and evolve.

A number of researchers [KLM+97] have begun working on approaches to this problem that allow programmers to express each of a system's aspects of concern in a separate and natural form, and then automatically combine those separate descriptions into a final executable form using automatic tools. These approaches have been called aspect-oriented programming (AOP).

In this workshop, rather than focussing on the idea of automatic weaver tools, a more general notion of AOP was adopted: AOP was regarded as a general concept or mechanism to solve the problem of modelling the different aspects of concern in a system. The purpose of the workshop was to bring together researchers and practitioners working in the area of AOP or related areas to discuss the current status of AOP research.

2 About the Workshop

The second workshop on *aspect-oriented programming* was organised by Cristina Videira Lopes, Gregor Kiczales, Kim Mens and Bedir Tekinerdogan on June 10 during the 11th European Conference on Object-Oriented Programming in Jyväskylä, Finland. (The first AOP workshop — the "AOP friends meeting" — was held at Xerox PARC in conjunction with OOPSLA'96.)

All participants were encouraged to submit a short position paper and the workshop was organised around the common tendencies detected in these position papers, such as:

1. What exactly are *aspects*? How can they be identified or characterised? [Meu97,MJV+97]
2. What is the difference between an aspect and a *component*? How do components and aspects interact? [HOT97,Lam97,Van97]
3. How to *weave*? (I.e. how to merge the base component program and the different aspect programs into a final executable form.) [Lam97]
4. Need for a theoretical foundation for AOP. [Meu97]
5. How to expand the use of aspects to other phases of the software development life-cycle: requirements, analysis, architecture, design, implementation, maintenance, ... [Aks97,HOT97,MJV+97,Mul97,Wer97]
6. What are the relationships or differences between AOP and other approaches or programming paradigms and especially between AOP, reflection, open implementations and meta-object protocols? (For example, is AOP better than a general framework like reflection?) [CES97,Meu97,DC97,MJV+97,Lam97]
7. Visual representations of AOP. (For example, visual presentation of relationships between components/aspects, graphical representations of aspects and aspect weaving, ...) [HOT97,Van97,Wer97]

8. Whereas the topics enumerated above are of a more general nature, many position papers mentioned specific concerns such as feedback on specific aspects and domains for AOP:
   - How to express the "coordination" aspect in concurrent OO? [HPMS97]
   - "Synchronisation" is not a single aspect but should be separated in several more specific aspects. [HNP97]
   - How to specify "failure detection" and "failure handling" in distributed OO using AOP? [Roy97]
9. Another important question which was not raised in any position paper is how to prove that AOP is good (i.e. better than existing approaches).

The goal of the workshop was not to find a definite answer to the above questions, but to use them as a general starting point for discussions. During the workshop, participants were encouraged to come up with other relevant questions and issues. The main purpose of the workshop was to identify the good questions that can lead to a characterisation of what AOP is and is not about.

3 About the Participants

During the warm up session, the participants were asked to introduce themselves, give a short summary of their position statement and optionally raise some questions for discussion.

- Many participants suggested new domains where AOP might prove useful such as distribution and mobility, automatic failure detection, coordination, synchronisation, load balancing, ...
- A number of other participants mentioned their interest in reuse and evolution issues, and the relation between AOP and current research issues in the reuse world.
- The relationship between AOP and composition mechanisms was also deemed interesting by many people.
- Peter Werner and some others stated that apart from aspect-oriented "programming", also aspect-oriented "modelling" and aspect-oriented "design" are important. This remark is strongly related to common tendency 5.
- Some people were a little sceptic. Sathoshi Matsuoka wondered whether AOP languages are needed at all, or whether we can suffice with conventional OO techniques and existing computational models. Wolfgang De Meuter mentioned that it might be possible to model AOP by means of meta-level programming (rather than considering meta-level programming a subset of AOP).
- Most of the participants talked about AOP in a general sense. Mira Mezini's statement that "AOP is not a programming paradigm but a design framework for separation of concerns" reflected this general understanding about AOP during the workshop.

4 Selected Presentations

Some of the submitted position papers raised more interest than others, especially the ones that made more general observations about AOP. Five authors were selected to present their position statement in more detail.

4.1 Aspects Should Not Die

Bert Robben [MJV+97] starts out with a discussion on the nature of aspects:

1. All aspects should be considered equally important within the context of a single application. This encourages aspectual decomposition from the very beginning.
2. What is the domain of aspects? Do they only deal with run-time properties (such as performance enhancement) or also with elements of the problem domain?
3. What is the appropriate abstraction level at which to describe aspects? Using separate high level declarative aspect description languages seems more appropriate than using the same language as the component language.
4. How and when do aspects show up during the development cycle? During which stages are they manifest as separate entities? Up to which point are the aspects orthogonal?

Next, the above mentioned issues were used to compare AOP with related approaches such as meta-object protocols and open implementations. As an example, consider Table 1 which compares the manifestation of aspects during different development stages for each of the approaches.

Table 1. Manifestation of Aspects

Development Stage	Traditional	MOP, OI	AOP now	AOP tomorrow
modelling	implicit	explicit	explicit	explicit
description	hard-coded	some aspects	explicit	explicit
run-time	fuzzy	some aspects	weaved	some explicit, some weaved

As can be seen from the last column in Table 1, the same issues were also used as a basis for identifying some possible future trends in AOP research. For example, it was argued that aspects (or at least some of them) must survive in the executable code if dynamic behaviour is to be supported.

In current AOP, aspects are only explicit until weave-time. An aspect weaver takes the aspect descriptions and tightly interconnects them with the application's functionality. In tomorrow's AOP at least some of the aspects (e.g., load balancing) should survive at run-time, to ensure maximal flexibility and to allow an aspect to adapt itself based on execution time information. Aspects that can be statically dealt with (e.g., synchronisation) can still be woven as before.

4.2 A Comparison of AOP-related Approaches

Krzysztof Czarnecki [CES97] discussed some problems with currently existing object-oriented technologies based on a comparison of different approaches from 3 different research communities:

1. Software Reuse,
2. Formal Transformational Development (generative programming),
3. OO and Adaptability Research.

The approaches were compared using the following criteria:

- Is the configuration time static or dynamic? (I.e. construction time or run time?)
- Which kinds of design knowledge can be expressed?
- Which kinds of optimisations are possible? (Global versus local and static versus dynamic.)
- What coordination mechanisms are used? (In other words, what are the *join points*?)
- Which concerns can be addressed?

He also argued that, to some extent, all discussed approaches (including AOP) strive towards reaching the same common goals:

- obtaining a (more) direct correspondence between requirements and code segments;
- raising the abstraction level;
- improving adaptability, extensibility and reusability;
- achieving a "complete" separation of concerns;
- achieving a "complete" separation of concerns *and* at the same time achieving high performance.

The important contribution of AOP could be to make these ideas practicable in industry.

4.3 Issues in Aspect-Oriented Software Development

Mehmet Aksit [Aks97] argued that aspect-oriented programming must be considered in a broader context. It is common practice to decompose software development activities into various phases, like requirements specification, domain analysis, architecture definition, design, implementation and maintenance. These phases are defined based on the viewpoints of the software engineer (analysis deals with what to do, design with how to do it, etc.). Since the concerns adressed in each of these phases have a major impact on the final structure and quality of software, they must be recognised as aspects. Going from one phase to another is then actually an aspect weaving process.

We can *identify* aspects by considering software development as a problem solving activity. The problem is typically represented by the requirement specification for which we try to find (software) solutions. The solutions are inherently

defined by the requirement specification and the domain knowledge. Aspects and aspect weaving processes have to be derived from the canonical models of these solutions. So clearly, aspect identification should start in the requirements specification and domain analysis phases, and not in the implementation phase. Aspects identified in the upper level phases of software development will have impact on the following phases. However, each subsequent phase may add new aspects and/or refine the existing aspects.

From the perspective of adaptability and reusability, mapping these solution techniques to the conventional object-oriented language mechanisms performs unsatisfactorily. Especially, multiple views, synchronisation and conditionally changing behaviour cannot be implemented well. Inheritance-based solutions perform better, but they cannot implement dynamically changing behaviour. The conventional object-oriented model requires 3 to 5 times more method implementations than the ideal case. The composition-filters model provides almost an ideal solution. In the composition-filters approach, the basic behaviour is implemented by using any programming language, and the additional aspects are defined in the filters. However, the composition-filters model is not capable of expressing aspects and weaving process at the design-level. Therefore, new techniques must be defined for design-level aspects and aspect weaving processes. Important characteristics of design level aspects are that they are mostly based on uncertain factors and that they are conflicting, context-dependent and non-deterministic.

4.4 Monads as a Theoretical Foundation for AOP

One of the reviewers qualified Wolfgang De Meuter's position paper [Meu97] as "an interesting beginning to the semantics of AOP and AOP in a functional programming setting".

The author proposes a theoretical foundation for AOP, based on the notion of *monads* known from functional programming. Aspects can be thought of as monad transformers, the base component program as a monadic style program and aspect weaving as monad transformation. The join points correspond to the "bind" operation on monads in combination with the other monadic operations.

As an experiment, De Meuter implemented a Fibonacci method to which the "aspects" of result caching and concurrent computation were added in a monadic way. These experiments indicate that the monad concept might be a very good candidate to give a formal semantics to AOP languages.

Besides providing a theoretical foundation for AOP in general, the proposed theory could also be regarded as a way of introducing AOP in the functional programming paradigm. An aspect-oriented program in a functional programming language would be nothing more than a monadic style program.

4.5 The Interaction of Components and Aspects

One of the realities of AOP is that aspect code and component code interact. It is this interaction that makes weavers necessary and that makes AOP interesting.

Different AOP approaches can be classified in terms of what the join points are and how the components and aspects interact.

John Lamping [Lam97] made a first classification of AOP approaches based on how the aspect behaviour and component behaviour are combined. In other words, how does the aspect code and the base code fit together? He distinguished between 3 ways of combining aspect and component behaviour, and gave some examples for each of them.

Juxtapose. Interleave doing aspect and component behaviour. In other words, the structure of the woven code looks basically like the base code, with aspect code added at the join points. (E.g., Iguana, Oz, composition filters, monads, coordination.)

Merge. As opposed to juxtaposition, when merging, a combination of aspect and component descriptions can be merged into a single action. (See the numerical code example in [KLM+97])

Fuse. An example of fusing can be found in the image processing example of [KLM+97] (loop fusion), where a single action is a combination of both aspect and several component level descriptions. In other words, several component level and aspect level descriptions can be fused into one single action.

A second classification can be made based on what kind of contextual information is needed. What kind of information about the context of execution of the component code is needed to choose the aspect behaviour? (I.e., what kind of information *not* maintained by the aspect code is needed by it?) Again, several kinds of contextual information can be distinguished:

Local. Composition filters and Oz only use information that is lexically nearby. In the image processing example, on the contrary, non-local information is needed: in order for a loop fusion aspect to fuse two loops it must examine two loops from the component code, which may potentially not even be adjacent in the component code.

"History". E.g., composition filters.

"Future". E.g., image processing, monads.

"Simultaneous". E.g., Iguana, coordination.

5 Afternoon Session — General Discussion

During the afternoon, about 40 participants joined in a plenary discussion of the following topics:

1. How do aspects and components interact?
2. Is aspect-orientation bound to object-oriented programming?
3. Are general purpose aspect languages possible or useful?
4. Can current technologies be used for AOP or do we need yet another technical development? (What existing techniques for manipulating computations exist?)
5. How can aspects be identified?
6. Which concerns does or should AOP separate?
7. Which problems can AOP solve? What are the hard problems?

5.1 Interaction of Aspects and Components

The first discussion was a continuation of John Lamping's presentation 4.5.

Mehmet Aksit did not completely agree with the classification that composition filters can depend on local contextual information only, as global objects can also be composed locally. Furthermore, composition filters do not only allow juxtaposition of aspect and component behaviour, but also merging. If the aspect code can be inferred in the compiler it can be merged with the base code.

There was also an undecided discussion on whether MOP should be considered an example of juxtaposition or merging.

5.2 Is Aspect-Orientation Bound to Object-Oriented Programming?

Now let us turn to the question of whether aspect-orientation is bound to object-oriented programming. In fact, this question can be decomposed in two questions:

1. Is aspect-orientation bound to object-orientation?
2. Is aspect-orientation bound to programming?

From the conceptual viewpoint, it is generally agreed that the object-oriented paradigm can model real world entities in a neat and understandable way. However, object-orientation lacks in adequately solving the problems which arise when different concerns, like real-time, synchronisation and coordination need to be composed together (and with the real world entities).

The reason for these modelling problems is the lack of expressive solution models and the lack of adequate composition mechanisms for such concerns. We cannot easily map the cross-cutting concerns to concepts of the conventional object model and we are not able to compose them in an orderly way. Aspect-orientation arose from the need to solve these modelling problems and accordingly addresses two basic issues. Firstly, how should we separate the real world concerns? Secondly, how should we compose these concerns at compile-time and at run-time?

Aspect-orientation advocates the use of expressive models for both components and concerns. By mapping real world concerns to aspects the cross-cutting behaviour of the different concerns will be eliminated and accordingly software systems will be better maintainable and adaptable. Clearly, like object-orientation is not bound to programming only, aspect-orientation can also be considered as a modelling technique and a mechanism which applies to all the phases of the entire software development cycle. Consequently, we can speak of aspect-oriented analysis (AOA), aspect-oriented design (AOD), aspect-oriented programming (AOP).

Further, we can state that aspect-orientation is not bound to object-orientation only. All existing programming paradigms like procedural, functional, logical and object-oriented paradigm provide models to express real world entities. In aspect-orientation conceptually an explicit distinction is made between *aspect languages* with which cross-cutting concerns are expressed, and *component languages* with which real world entities and the basic computation functionality are

expressed. Each aspect should be expressed in its own natural language. As such, in addition to the basic computation language we may for example have specific aspect languages for concurrency, real-time and coordination concerns. Conventional languages may equally both be used as component languages and aspect languages. The component and aspect languages might even be the same. The choice of the language inherently depends on the problem and additional context parameters. The fundamental point however is that aspect orientation intrinsically advocates the use of those languages — possibly from different paradigms — that are most natural for the task at hand. In this sense we could say that aspect orientation is rather independent of the existing paradigms.

5.3 General Purpose Aspect Languages

Would it be *possible* to get an aspect language that is general purpose to the same degree that an OO language is general purpose? The advantage of using a general purpose language such as, for example, C++ is that everyone knows it and can understand it. It is a common way of expressing the semantics and freezes the patterns of usage that programmers are used to.

But do we *want* general purpose aspect languages or do we prefer many different aspect languages? In general there is a trade-off between using a single general purpose or many specific aspect languages. Mehmet Aksit gave the example of composition languages. If all composition filters are written in the same language as the base program, the weaving is much easier. But now suppose you want to deal with real-time filters. First the composition language will need to be extended to deal with real-time aspects, but all the rest will become more difficult as well. With separate languages you only need one extra language in which to describe the real-time aspects.

There are some other advantages to using many different aspect languages rather than a single general purpose language. When using appropriate aspect languages the aspect code will be more concise and easier to understand, and will limit the programmer to mess up. General purpose languages for aspects are not good because they do not allow to describe the aspects at the right abstraction level. A related motivation for using different languages is that lots of aspects have to do with control flow. This can be modelled very well by means of constraints but poorly by imperative code. Hence the subject matter of aspects is different than the subject matter of components and it is probably better to use different languages.

Having different aspect languages is neither necessary nor sufficient for AOP. Indeed, in some cases it may be convenient to write both the aspects and the component program in the same language, whereas in other cases using different aspect languages seems more advantageous.

But if you use a number of aspect languages, eventually they will need to be translated into a single language. Which facilities or features does this language need to provide? Current AOP languages do not seem to require anything special, as long as the output language is low level enough. But can we have a high enough

level output language? Is it possible to make a general purpose intermediate aspect language so that it is easy to translate into any other language?

And even if you do not want a general purpose aspect language, can we provide general aspects weavers, or do we need domain specific weavers?

As a final remark, it should be remembered that the "generality" of aspect languages and aspect weavers will always apply only to some extent. Therefore, it might be better to talk about the scope of generality of an aspect language or weaver.

5.4 Reflection versus AOP

In the context of discussion topic 4 there was some discussion on how much help can be expected from reflective techniques. Reflection certainly seems to be a sufficient mechanism for AOP, but many regard it as too powerful: anything can be done with it, including AOP. When using reflection, will the aspect-oriented program be safe or efficient enough? Is reflection required to make the program adaptable enough? More research is needed here.

Someone argued that reflection is too powerful because of its focus on mechanisms rather than on the structure of the meta-level. In other words, what is missing to constrain the reflective power is a composition methodology at the meta-level.

5.5 Identification of Aspects

How can aspects be identified? What aspects should we be looking for? What other domains are there?

Someone suggested that when a problem is decomposed in subproblems, every subproblem can be considered as an aspect. However, this *cannot* be the case as aspects are not packaged in one component but come out of the interactions between components. (Kiczales mentioned that this is also the reason why subjectivity does not feel like aspects.) In fact, this is precisely a characterisation of the difference between aspects and components. Components are those things you obtain when breaking something in pieces of functionality, whereas aspects are those things that remain and are difficult to describe locally with respect to those components. However, it is possible that with another choice of components some aspects become components and vice versa.

If we want to identify aspects, it would be a great help to have explicit software entities that map onto the aspects. For example, in the functional programming paradigm, programs can either be structured according to the values they consume or according to the computations they consume. The latter style of programming is called *monadic programming* [Meu97]. These two styles of functional programming seem to correspond to component programs and aspect programs, respectively. Furthermore, AOP is about having both kinds of programming styles simultaneously. This is the same in monadic programming where you still need the component way of programming as well. So there seems

to be a close correspondence between AOP and monadic style functional programming.

5.6 Separation of Concerns

With AOP we want to separate out different aspects at a more convenient abstraction level. Furthermore, we want to describe these aspects independent of the components in the base program and use weavers to avoid having to visit all components. AOP typically tries to separate some of the concerns that component-based technologies are not good in decomposing. But AOP should not be seen as a *complete* separation of concerns: the aspects still have to do with the components. You still have to look at the components, but not at everything, only at the things you want to see. Achieving a complete separation is not only hard, it is not even a goal: if you would have complete separation, the things that are separated would not be part of the same system anyway. But the question remains how much we want things to be separated.

5.7 Which Problems to Solve with AOP?

The last discussion topic regarded application domains for AOP. Which problems can AOP solve? What are the hard problems? Due to a lack of time only one interesting new application domain for AOP was suggested.

No position paper mentioned the use of AOP for writing web-software, where many components are being updated and changed at different rates. Although AOP is not an approach specific to this area, one might wonder whether an AOP approach could be of value here. One important issue when writing web-software is to be able to control the *interaction* between the objects. (You want to control the interaction between the objects rather than the objects themselves, because you do not own the objects.) At first glance, AOP seems a useful approach because it is good in addressing such a non-local thing.

6 Concluding Remarks

AOP defines a new concept, called aspect, that enables us to talk about an important new kind of modular unit in system designs and implementations. Aspects are intended to work together with traditional notions of components, including modules, objects, API's and the like, but typically address concerns that cut-across groups of these components. Aspect-oriented programming is a style of programming in which aspects and their interactions with components are clearly identified. Aspect-oriented programming can include specific aspect languages to program the aspects, or can be done with existing programming languages and coding idioms that make the aspects more clear.

With this as our background, it is clear that a lot of work remains to be done. Some of the key issues that were addressed during the workshop are summarised below:

Need for more technical research. Whereas there seems to be a common intuition on what AOP actually is, it is equally clear that the technical precision behind that intuition needs to be worked out. For that, a complete catalogue should be made of the precise technical problems that need to be solved.

Need for AO*. From the discussion on the use of aspects throughout the software life-cycle we can deduce that there is not only a need for Aspect-Oriented Programming (AOP), but also for Aspect-Oriented Analysis (AOA), Aspect-Oriented Design (AOD), Aspect-Oriented Modelling (AOM), and so on. AOP should be scalable to these domains.

Need for AOP metrics. To justify the claim that AOP actually makes building real software easier, measurable results for AOP are needed. (Metrics of code tangling.)

Can existing technologies be used for AOP? During the workshop there was a lot of discussion about the relation between reflection and AOP. The key question here is how much of the technology that is needed for AOP (or AO*) is already available. Is there really a need for a new technical development, or is (for example) reflection or meta-programming already sufficient?

Need for comparisons between AOP and related approaches. As people will want to know whether AOP is new or whether it is nothing more than a new name for an old thing, comparisons between AOP and related work (such as composition filters and subject-oriented programming) are important.

Need for a theoretical foundation. It is obvious that lots of research is needed on theoretical foundations for AOP, for example, monads.

Separation of concerns in AOP. It should be made clear what we set the goal of separation of concerns to be: what are the concepts we want to separate, and how much do we want them to be separated?

Is AOP bound to OO? This is only a rhetorical question as we clearly want AOP *not* to be bound to OO only. The right question to ask is what communities we should be talking to about this idea and for help with this idea.

Other future work. Apart from the general considerations above, some more specific questions and topics to be investigated are:

- What is the domain of aspects?
- How to identify aspects?
- Aspect description languages.
- Orthogonality of aspects.
- Translation techniques.
- How to weave? What should a weaver do?
- Run-time versus earlier time aspects.
- How to specify join points?
- General purpose or domain specific AOP?
- Aspects applied to existing libraries.
- How to deal with evolution?
- Visual representations of AOP.

7 Acknowledgements

We express our gratitude to the reviewers of the submitted position papers: Mehmet Aksit, Lodewijk Bergmans, Pierre Cointe, Theo D'Hondt, Karl Lieberherr, Carine Lucas, Calton Pu and Michael VanHilst.

We would also like to thank the participants who sent in a position paper as well as all other participants who joined the discussions in the afternoon sessions of the workshop: Vito Baggiolini, Per Brand, Vinny Cahill, John Dempsey, Ulrich Eisenecker, Marc Evers, Bjorn Freeman-Benson, William Harrison, Juan Hernández, David Holmes, Wouter Joosen, J. S. Madsen, Francesco Marcelloni, Satoshi Matsuoka, Frank Matthijs, Jürgen Müller, Juan Murillo, James Noble, Harold Ossher, Michael Papathomas, John Potter, Fernando Sánchez, Patrick Steyaert, Peri Tarr, Kresten Krab Thorup, Klaas van den Berg, Pim van den Broek, Willem van den Ende, Bart Vanhaute, Michael VanHilst, Peter Van Roy, Pierre Verbaeten and Peter Werner. Special thanks to Mehmet Aksit, Bert Robben, Krzysztof Czarnecki, Wolfgang De Meuter and John Lamping who presented their position statement at the workshop as well.

References

[Aks97] Mehmet Aksit. Issues in aspect-oriented software development. Position paper at the ECOOP'97 workshop on Aspect-Oriented Programming, 1997.

[CES97] Krzysztof Czarnecki, Ulrich W. Eisenecker, and Patrick Steyaert. Beyond objects: Generative programming. Position paper at the ECOOP'97 workshop on Aspect-Oriented Programming, 1997.

[DC97] John Dempsey and Vinny Cahill. Aspects of system support for distributed computing. Position paper at the ECOOP'97 workshop on Aspect-Oriented Programming, 1997.

[HNP97] David Holmes, James Noble, and John Potter. Aspects of synchronisation. Position paper at the ECOOP'97 workshop on Aspect-Oriented Programming, 1997.

[HOT97] William Harrison, Harold Ossher, and Peri Tarr. The beginnings of a graphical environment for subject-oriented programming. Position paper at the ECOOP'97 workshop on Aspect-Oriented Programming, 1997.

[HPMS97] Juan Hernandez, Michael Papathomas, Juan M. Murilli, and Fernando Sanchez. Coordinating concurrent objects: How to deal with the coordination aspect? Position paper at the ECOOP'97 workshop on Aspect-Oriented Programming, 1997.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97 Proceedings, Lecture Notes in Computer Science, Springer-Verlag*, pages 220–242, 1997.

[Lam97] John Lamping. The interaction of components and aspects. Position paper at the ECOOP'97 workshop on Aspect-Oriented Programming, 1997.

[Meu97] Wolfgang De Meuter. Monads as a theoretical foundation for aop. Position paper at the ECOOP'97 workshop on Aspect-Oriented Programming, 1997.

[MJV+97] Frank Matthijs, Wouter Joosen, Bart Vanhaute, Bert Robben, and Pierre Verbaeten. Aspects should not die. Position paper at the ECOOP'97 workshop on Aspect-Oriented Programming, 1997.

[Mul97] Jurgen K. Muller. Aspect-design in the building-block method. Position paper at the ECOOP'97 workshop on Aspect-Oriented Programming, 1997.

[Roy97] Peter Van Roy. Using mobility to make transparent distribution practical. Position paper at the ECOOP'97 workshop on Aspect-Oriented Programming, 1997.

[Van97] Michael VanHilst. Subcomponent decomposition as a form of aspect-oriented programming. Position paper at the ECOOP'97 workshop on Aspect-Oriented Programming, 1997.

[Wer97] Peter Werner. Position statement submitted to the ECOOP'97 workshop on Aspect-Oriented Programming, 1997.

All position papers submitted to the workshop are available on the web-site `http://wwwtrese.cs.utwente.nl/aop-ecoop97/`