Aspect-Oriented Programming

Lodewijk Bergmans[1] and Cristina Videira Lopes[2] (Editors)

[1] University of Twente, The Netherlands
bergmans@cs.utwente.nl
[2] Xerox Palo Alto Research Center, Palo Alto, CA
lopes@parc.xerox.com
http://trese.cs.utwente.nl/aop-ecoop99/

Abstract. Aspect-oriented programming is a promising idea that can improve the quality of software by reduce the problem of code tangling and improving the separation of concerns. At ECOOP'97, the first AOP workshop brought together a number of researchers interested in aspect-orientation. At ECOOP'98, during the second AOP workshop the participants reported on progress in some research topics and raised more issues that were further discussed.
This year, the ideas and concepts of AOP have been spread and adopted more widely, and, accordingly, the workshop received many submissions covering areas from design and application of aspects to design and implementation of aspect languages.

Workshop organisers: Cristina Lopes, Andrew Black, Elizabeth Kendall, Mehmet Aksit, Lodewijk Bergmans

This report received contributions from (in alphabetical order): L. Blair, K. Böllert, S. Clarke, C. Constantinides, Y. Gil, M. D'Hondt, L. Kendall, G. Kiczales, J. Knudsen, R. Lämmel, J. Lamping, K. Mehner, L. Pazzi, J. Pryor, J. Seinturier, M. Skipper, M. Südholt, S. Thompson, I. Welch

1. Introduction

Many systems have properties that do not necessarily align with the system's functional components. Failure handling, persistence, communication, replication, co-ordination, memory management, real-time constraints, etc., are aspects of a system's behaviour that tend to cut-across groups of functional components. While they can be thought about and analysed relatively separately from the basic functionality, programming them using current component-oriented languages results in spreading the aspect code through many components. 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

A. Moreira and S. Demeyer (Eds.): ECOOP'99 Workshops, LNCS 1743, pp. 288–313, 1999.

components less reusable. In short, it makes the source code difficult to develop, understand and evolve.

Aspect-oriented programming is a promising idea that could reduce the problem of code tangling, and therefore improve the quality of software. At ECOOP'97, the first AOP workshop brought together a number of researchers interested in aspect-orientation. At ECOOP'98, during the second AOP workshop the participants reported on progress in some research topics and raised more issues that were further discussed.

This year, the ideas and concepts of AOP have been spread and adopted more widely. Accordingly, the workshop received 26 submissions, of which 23 were accepted, covering areas from design and application of aspects to design and implementation of aspect languages.

The program consisted of the following five sessions:

1. An invited talk by John Lamping
2. Applications Session
3. Specification and Design Session
4. Implementation Session
5. Aspect Language Designs Session
6. Wrap-up Session

The first five sessions consisted of a number of presentations followed by discussions. Excerpts of these discussions have been included in this report in the following forms:

!: [Mrs. X] A remark or suggestion made by Mrs. X.

Q: [Mr. Y] A question by Mr.Y to the presenter.

A: An answer, usually by the presenter, if not denoted otherwise.

The last session was a regulated interactive session intended to obtain suggestions from all participants as to what they find important and relevant about AOP.

This workshop report is organised as follows: the following six sections provide a description of each of the sessions described above. Finally, there is a list of participants and their submissions, as well as the collected references.

Note that these excerpts have been reconstructed afterwards from notes that were made, and provide only a short summary of what happened during the workshop. There were some email discussions before the workshop - they can be found in the workshop's web page. Also, for an in-depth view of the works that were submitted, we refer the reader to the collection of workshop papers that can be found in the workshop's web page.

2. Invited Talk (John Lamping)

John Lamping has been working at XEROX PARC on Aspect-Oriented Programming from the very start, he was invited to open the workshop with a talk about his latest insights and ideas. The following is a summary of his talk.

The presentation revolved around the picture in Fig. 1. In object oriented programming, the structure of the basic operations, the messages that objects

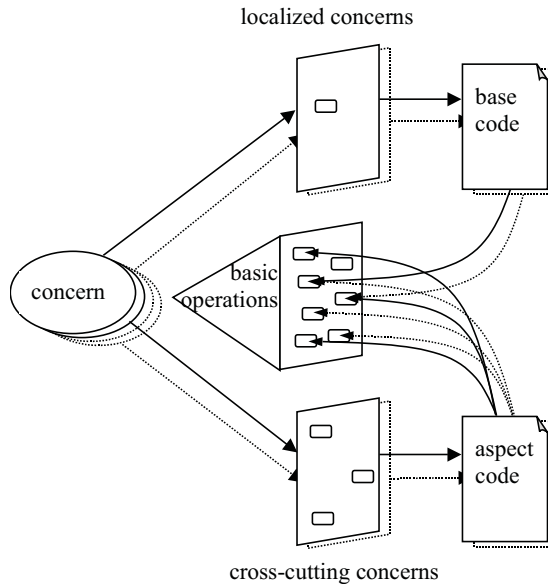


Fig. 1. John Lamping's picture.

can respond to, aligns with the structure of the code, the class files and methods. But in aspect oriented programming the relationship is more subtle, and more interesting.

The designer of an aspect-oriented program identifies the domain and implementation concerns their program must meet, and they also decide on the basic operations the program will be expressed in terms of. For AspectJ, the basic operations are the messages that the program will operate in terms of. These messages play as central a role in AspectJ programs as they do in ordinary OO programs. In particular, the choice of the basic operations determines which concerns will be localised, pertaining to one or a few basic operations, and which will cross-cut the basic operations, pertaining to a number of them. A good choice of basic operations will lead to as many concerns being localised as possible, but aspect oriented programming recognises that it is typically impossible to localise all concerns.

Those concerns that are localised are addressed by base code, while those concerns that cross-cut are addressed by aspect code. Both base code and aspect code contribute to the specification of what should happen when the basic operations are executed.

In summary, the basic operations play a central role in aspect oriented programming, both by determining which concerns cross-cut, and by serving as the common ground where the effect of base and aspect code meets.

Questions and Discussion

!: [Yossi Gil] Design decision about the difference between concerns and base operations are still difficult.
!: [Dominick Lutz] Whatever formalism you adopt, you will always have to make design decisions about how to decompose a system.
Q: [Lodewijk Bergmans] How can you express aspects of other aspects?
A: All aspects can be expressed in terms of the base operations, it is even possible in this way to model aspects of aspects.

3. Applications

This session, about the application of the concepts of AOP to various problems was chaired by Liz Kendall. The presenters of this session were asked beforehand to address the following questions and issues:

- What is the significance of the presented application.
- What about alternative solutions/approaches?
- What are the benefits and drawbacks of AOP solution?
- How has the development of this application progressed your knowledge and understanding of AOP?
- Share your experience with AOP
- How does AOP influence specification and design?
- Will this approach scale to larger applications?
- What is the next step/future work?

3.1. An Aspect Language for Robust Programming (Mario Südholt)

Presentation of the position paper submitted by Pascal Fradet and Mario Südholt

Topics

In this talk a semantically based robustness aspect for numerical programs was presented. The approach provides a specialised aspect language for the specification of the exceptional value domains of component programs. The specifications define program transformations that are used by the weaver to transform the component program into a woven program that provably does not perform calculations based on exceptional values.

Lessons Learned/Conclusions

A very specialised aspect language is useful to describe the robustness aspect as declaratively as possible. In the case of the robustness aspect, statement-level join points are not fine-grained enough. General pattern-based join point definitions

have proven to provide adequate flexibility while retaining declarativeness of aspect definitions.

The talk advocated a three-level approach with a clean separation between levels: the aspect language is used to write aspect programs at the user level, the aspect language definition is done at a lower-level (linguistic level which is inaccessible to the user), aspect weaving is done at a generic implementation level supporting tool support.

The approach features a formal (with respect to syntax and semantics) definition of aspects and aspect weavers. By keeping the weaver simple important properties of the woven program can be proven based on the aspect only, i.e. (almost) without knowing the weaver.

Open Issues/Future Work

Since one of the main features of the approach is its formal definition, the question which properties, i.e. aspects, can be integrated must be investigated.

Current work is done in two main areas:

1. Investigating general properties of the formal framework, in particular with respect to the integration of new aspects (properties).
2. Developing a debugging and a security aspect

Questions and Discussion

Q: [Dominick Lutz]: How can you be sure that your pattern matches in the right place(s)
A: The specification allows syntactical patterns of any desired granularity.

3.2. JST: An Object Synchronisation Aspect for Java (Lionel Seinturier)

Presentation of the position paper submitted by Lionel Seinturier

In his talk, Lionel Seinturier presented JST, an aspect weaver for the Java language. JST addresses two aspects: object synchronisation and observation of a distributed CORBA run. The idea is to provide a tool (i.e. a so called aspect weaver) that allows to separate the code related to these two issues from the functional code of a distributed and concurrent CORBA/Java program. The synchronisation aspect is associated to a language with a statechart-like syntax. It wraps base level object and synchronises method calls before delivering them. The join point is the base level class interface. The observation aspect of JST is associated with some annotations of the base level programs. They point out the elements (methods, variables) that need to be traced. The data collected during a distributed run can then be used to perform some post-mortem profiling. This task is based on an extension of the Lamport causality relation. The extension

proposed by Lionel Seinturier adds three sources of order (synchronous method calls, synchronised methods, and read/write dependencies on shared variables) to the two sources (local ordering of events, and asynchronous communications) considered by the Lamport relation. JST is implemented with OpenJava (a compile-time MOP for Java) and ORBacus (a CORBA ORB). A first version of JST can be downloaded from Lionel Seinturier home page at the following URL: `http://www-src.lip6.fr/homepages/Lionel.Seinturier/JST`

Two conclusions can be drawn from Lionel Seinturier's work. First, reflective languages such as OpenJava provide an useful help in implementing aspect weavers. According to him, a lot of code should have been rewritten if OpenJava hadn't been used for JST. Second, in his opinion, the design of dedicated languages for each aspect leads to a better understanding of the domains involved in the program and of the weaving process. Once the join point between the aspect languages and the base language has been understood, the design of behaviours for each aspect becomes easier.

One of the open issues with AOP lies in the debug process. Indeed, once a program has been woven, the relation between the base level and the aspect level code may not be so clear. Some work still need to be done to address this problem. Finally, according to Lionel Seinturier, another big open issue that remains to be addressed is to know whether AOP scales well and if it can be applied to large scale applications where complex aspects such as fault tolerance, replication or mobility are involved.

Questions and Discussion

Q: [Gregor Kiczales] Can you synchronise multiple instances of multiple classes?
A: [Seinturier] No. This kind of synchronisation has to be done by designing different synchronisation behaviours and associating each behaviour to a particular class.

3.3. Is Domain Knowledge an Aspect? (Maja D'Hondt)

Presentation of the position paper submitted by Maja D'Hondt and Theo D'Hondt

Programs are a combination of domain knowledge and algorithms. Moreover, the real-world example accompanying this talk shows that the first crosscuts the second. The benefits of factoring out domain knowledge are clear: the programming process becomes less complex and both the domain knowledge and the algorithm can evolve independently from one another.

As a first and obvious solution, the presentation pointed to the classical techniques of object-orientation, such as delegation, subclassing and the use of design patterns to factor out domain knowledge. These techniques, however, cannot be applied in all cases. For example, domain knowledge that evolves can sometimes force the addition of a new parameter in the original algorithm. In order to avoid this, this talk focused on aspect-oriented programming as a possible solution.

The exploration and development of a programming environment that supports the separation of domain knowledge from algorithms at coding time, but that weaves the two at compile time or at run time, remains further work. Nevertheless, some initial and successful experiments were mentioned in the talk, consisting of a language symbiosis between Prolog for representing the domain knowledge, and Smalltalk for the implementation of the algorithm.

Questions and Discussion

Q: [Jørgen Knudsen] Why not the OO solution: force the separation through different languages
A: [Theo D'Hondt]: this work is driven by the goal of reusing AI domain knowledge technology and adding it to conventional programming
Q: [Mario Südholt]: if you factor out the heuristics from the algorithm, doesn't that clutter the understanding and optimisation of the algorithm?
A: No, at least in this case rather the reverse, since all optimisations can be expressed as constraints that are derived directly from the domain knowledge.

3.4. Aspect-Oriented Programming for Role Models (Elizabeth A. Kendall)

Presentation of the position paper submitted by Elizabeth A. Kendall

The Role Object with the Decorator pattern has been proposed as the best support for role models in standard object-oriented languages, such as Java. However, the Role Object design has three major drawbacks: object schizophrenia; interface bloat or, alternatively, down-casting; and no support for role composition.

The hybrid approach for AOP-based role model implementation presented by Liz Kendall places role behaviour in a combination of introduce weaves that are added to the core class and advise weaves that are added to the core instances. That is, an aspect both introduces the interface to the core class and then advises or adds the role specific behaviour to the core instance. Further, role relationships and role context reside in the aspect instance to easily support role multiplicity.

Liz Kendall claimed the following benefits of this hybrid approach to role aspects:

- Interface maintenance: The class' own intrinsic interface is not bloated with every potential role. However, the extrinsic behaviour is also accessible without down-casting.
- Object schizophrenia: Most of the role specific behaviour resides in the object; only role relationships and role context reside in the aspect.

She had also investigated Glue Aspects, where roles are represented by objects, and aspects integrate a Core object to the role(s) that it plays. This approach has the following benefits and drawbacks:

- Independent Core and Role hierarchies: Any Core object can play a given Role if the appropriate Glue Aspect is provided. This is the major advantage of this design.
- Interface maintenance: The role specific interfaces are introduced to the Core objects in a modular fashion. This is also true in the hybrid approach.
- Role multiplicity: The glue aspect design provides support for role multiplicity as roles can be indexed by context. This is also true in the hybrid approach.
- Object schizophrenia: The Role and Core objects are independent, so the Glue Aspects have to encode and manage all integration. The hybrid approach is superior in this area, and glue aspects should only be employed when there are only minimal dependencies between Role and Core objects.
- Additional level of components: The major drawback of this design is that it requires three levels of components

The presentation included illustrative examples of these two approaches for AOP-based role model implementation.

3.5. Aspect-Orientated Process Engineering (Simon Thompson)

Presentation of the position paper submitted by Simon Thompson and Brian Odgers

This talk addressed the question of how process knowledge should be structured. The use of Aspect Orientation as an abstraction for capturing and using process knowledge was reported. A description of the method, ASOPE (Aspect Oriented Process Engineering) was given.

The ASOPE system implements a hierarchical planning system that iteratively specialises a plan template stored as an object. The specialisation is done by weaving context specific plan elements, stored as aspects, into the generic plan according to the ownership and execution requirements of the business goal that is to be achieved. This process is illustrated in Fig. 2.

The advantage of using an Aspect Orientated representation was reported to be that the process knowledge developed in one context could be decontextualised and reused in other contexts, and context specific knowledge could be captured and retained for reuse at a local level.

4. Specification and Design

This session was chaired by Crista Lopes. The two first presentations dealt with aspect-based specifications and the two subsequent presentations discussed adding the notion of aspects to the design process and notations.

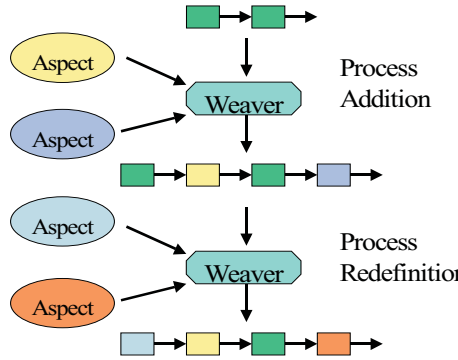


Fig. 2. Picture used in Simon Thompson's presentation. The hierarchical aspect weaving process used to specialise the generic plan.

4.1. A Tool Suite to Support Aspect-Oriented Specification (Lynne Blair)

Presentation of the position paper submitted by Lynne Blair and Gordon Blair

This talk presented an aspect-oriented style for the formal specification of systems, along with a supporting toolkit. The specification of a simple multimedia stream was considered and different aspects were identified, namely functional, non-functional and management aspects. If required, each aspect could be specified in a different formal language, for example using different notations for the functional (base) behaviour and the real-time or stochastic behaviour. Using a common underlying semantic model of timed automata, it was then possible to compose the aspects in order to perform analysis of either the interaction of aspects (c.f. feature interaction) or the overall system behaviour.

Composition in this approach was analogous to aspect-weaving in aspect-oriented programming and the multi-way synchronisation of events (either explicit or implicit) mirrored join-points. The composition process for the multimedia stream example was demonstrated using the Composer toolkit (see Fig. 3). The resulting behaviour was analysed using the tool's simulator and it was shown how temporal logic properties could be proved over the system by model checking.

Importantly, having checked the behaviour of the management aspects (i.e. the monitors and controllers) in the formal world, these aspects have been inserted directly into a running system using a reflective platform. Used this way, the management aspects described in the talk directly and dynamically monitor and control the behaviour of an audio stream.

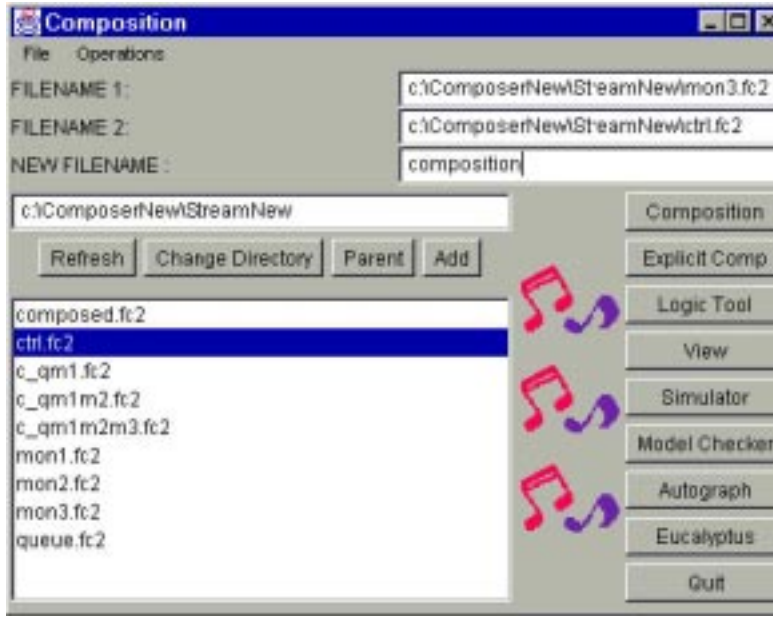


Fig. 3. A screen-dump from the Composer toolkit described and demonstrated by Lynne Blair.

4.2. Explicit Aspect Composition by Part-Whole State Charts (Luca Pazzi)

Presentation of the position paper submitted by Luca Pazzi

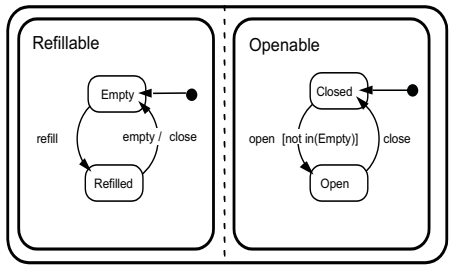
Luca Pazzi examined the issue of aspect weaving from two different approaches. As a working hypothesis, aspects were meant as specifications of generic and separable parts of the behaviour of an object. For example joining the aspects "being refillable" and "being breakable" may specify the full behaviour of a bottle, although both aspects are very generic and can be used to characterise any other object that is either refillable or breakable. Aspect specification means finding a suitable mechanism for both aspect specification and composition. The Statecharts formalism [Harel 77] was proposed to: a) specify aspects by separate, self-contained state Statecharts; b) compose aspects by the Statecharts AND composition mechanism.

It was showed that aspect composition could be achieved by two different approaches:

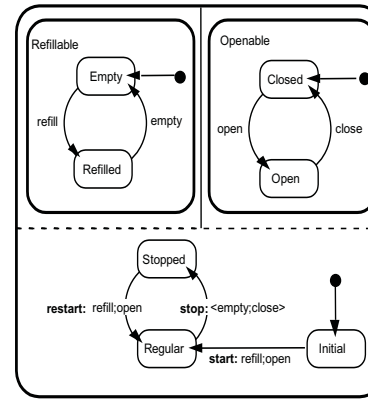
- Implicit weaving: A complex behaviour implicitly results by directly forwarding events from a Statechart to the another.
- Explicit weaving: A complex behaviour is explicitly depicted by a specific state machine whose states are compound states drawn from the Cartesian

product of the sets of states of the component machines, and state transitions are sequences of state transitions taken from the component machines.

Although the two approaches can be shown to be formally equivalent, they really differ from a pragmatic, cognitive and ontological point of view. In fact, it can be observed that the problem with the traditional implicit composition approach is that we have to add exogenous details to the specifications in order to make the global behaviour: this breaks the encapsulation of single aspects making them less reusable and understandable. On the other hand, explicit weaving, adopting a suitable formalism, such as Part-Whole Statecharts [Pazzi 97], allows to leave the original state machines untouched and to have both high level states and events denoting the resulting woven aspect. Fig. 4 shows these respective approaches.



a) Example of implicit weaving.



b) Example of explicit weaving.

Fig. 4. Pictures in Pazzi's presentation.

The implicit weaving of the two aspects "being refillable" and "being openable", by traditional Statecharts AND composition mechanism (denoted by the dotted line). Observe that the synchronisation of the two state machines requires to add exogenous details (underlined features) to the component Statecharts.

The explicit weaving of the two aspects "being refillable" and "being openable", by Part-Whole Statecharts [Pazzi 97]. The global woven aspect is denoted by an explicit state machine (representing the whole behaviour) whose state transitions denote high-level events and map to the lower-level events of the component state machines.

Questions and Discussion

A discussion emerged about implicit vs. explicit and automatic vs. manual join points. It was pointed out that these two are not the same. Implicit vs. explicit has to do with how much information is given to specify the join points. Automatic vs. manual has to do with how the composition is done.

4.3. Separating Concerns Throughout the Development Lifecycle (Siobhán Clarke)

Presentation of the position paper submitted by Siobhán Clarke, William Harrison, Harold Ossher, and Peri Tarr

Siobhán Clarke presented an extension to the notion of separating concerns at just the code level in a position paper called "Separating Concerns throughout the Development Lifecycle". She discussed the need to separate the same concerns across the lifecycle to ease both the initial development, and evolution throughout the software's life. With current object-oriented design methods, designs are caught in the middle of a significant structural misalignment between requirements and code.

This misalignment leads to the scattering of the design and code of individual requirements across multiple classes, and tangling, where individual classes in the design and code may address multiple requirements. Addressing this misalignment problem suggests that it must be possible to reify features within the object-oriented paradigm to permit encapsulation of feature concerns, as specified in the requirements, within designs and code.

In this talk, subject-oriented design was discussed, which is an outgrowth of the subject-oriented programming model. Subject-oriented design supports the decomposition of design models matching requirements specifications, and, is an outgrowth of the work on subject-oriented programming. Subject-oriented programming addressed misalignment and related problems at the code level, and as such, subject-oriented design is the bridge between how requirements are specified and how code can be successfully separated based on features.

The talk further described the subject-oriented design model. A full system design model is divided into design subjects, each of which encapsulates some concern in the object-oriented design. Composition relationships may be specified between design subjects, which specify which design elements in the different subjects correspond, how differences between the specifications of corresponding elements may be reconciled, and how they should be reconciled in a composition process.

4.4. Extending UML with Aspects: Aspect Support in the Design Phase (Junichi Suzuki)

Presentation of the position paper submitted by Junichi Suzuki and Yoshikazu Yamamoto

This talk addressed the aspect support in the design level while it has been focused mainly in the implementation/coding phase. The motivation is that Aspect-Oriented Programming (AOP) has been considered a promising abstraction principle to reduce the problem of code tangling and make software structure clean and configurable. Suzuki proposed an extension to the Unified Modeling Language (UML) to support aspects properly without breaking the existing

UML specification. This allows developers to recognise and understand aspects in the design phase explicitly. This is achieved mainly through the introduction of new stereotypes.

Also, he proposed an XML-based aspect description language, UXF/a, to achieve interchangeability of aspect model information between development tools such as CASE tools and aspect weavers. The goal of the work presented here is to facilitate aspect documentation and learning, and increase aspect reusability and interchangeability.

5. Implementation

This session was prepared by Andrew Black, who could not attend the workshop. In this session there were presented a wide range of approaches towards the implementation of aspects, targeted at different languages (e.g. Smalltalk, Java), and adopting different approaches (e.g. reflection, first class aspects, design pattern/framework based aspects, pre-processor).

5.1. A Reflective Architecture for the Support of Aspect-Oriented Programming in Smalltalk (Jane Pryor)

Presentation of the position paper submitted by Jane Pryor and Natalio Bastán
The presentation described a reflective architecture implemented in Smalltalk that permits the incorporation of aspects to an object-oriented system. The reflective mechanism is supported by the Luthier MOPs framework, where reflection is implemented by means of message interception (by wrapping methods), though other reflection and reification facilities can be added if necessary. The functional objects of the system reside at the base level and the aspects at the meta-level of the architecture. This permits not only a clean separation of concerns, but also permits the dynamic manipulation of aspects in a completely transparent fashion with no modification to the design or implementation of the functional part of the system. An Aspect Manager at the meta-level associates and activates the aspects at run-time, permitting a large degree of flexibility. In particular, cross-cutting is permitted between one or many classes and/or instances of base objects with one or many classes and/or instances of aspects, previously defined by a composition method which associates the aspect/s to the object methods.

In conclusion, this architecture which is now being implemented in Java, has the advantage of dynamic weaving and a very clean separation of the aspect and functional parts of a system. Additionally, the framework that supports it facilitates the incorporation of other facilities, such as the handling of constraints between aspects (the Aspect Manager is easily extended to incorporate additional functionality), and eventually aspects of aspects (through extra meta-levels for example). These considerations and the actual implementation of applications with the incorporation of different types of aspects are the next steps to be taken.

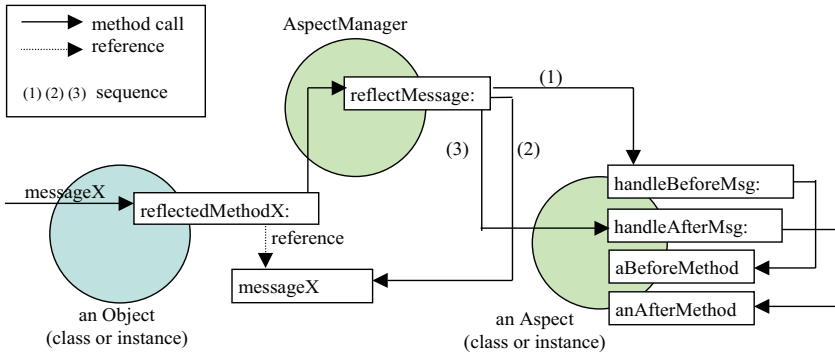


Fig. 5. Picture in Jane Pryor's presentation. Composition Mechanism: relationship among basic objects, the weaver and aspects.

As to the discussion of implementation issues, the pros and cons of static vs. dynamic weaving were mentioned, the problem with the latter mainly being a question of efficiency: for how much longer will this be an issue? In addition, the incorporation of constraints between aspects and the notion of aspects of aspects, should be issues to be considered in future proposals.

Questions and Discussion

Q: [Gregor Kiczales]: why aren't the before/after-method at the base-level?
A: This is possible, but it was not done for separating the aspect definitions from the base level.

5.2. On Weaving Aspects (Kai Böllert)

Presentation of the position paper submitted by Kai Böllert

Today development of aspects is done either in several special aspect languages or in one general-purpose aspect language (e.g., AspectJ). In contrast, Kai Böllert took the position that it may be better to use the same (object-oriented) programming language for writing both components (i.e., classes) and aspects. To illustrate how such an approach could be realised, Böllert showed in his talk the next version of the AOP/ST tool, which adds AOP extensions to the Smalltalk development environment VisualWorks.

The three most important benefits of not writing aspects in one or more special aspect languages are:

1. Significantly reduced implementation effort for AOP extensions, because no aspect language compiler needs to be implemented.

2. Easier integration of AOP into commercial development environments, because existing tools like browsers and debuggers can be reused for aspect development.
3. Facilitates adoption of AOP, because developers do not need to learn a new language.

Regarding the implementation of Aspect Weavers, Kai Böllert pointed out an open research question: is it possible for an Aspect Weaver to handle relationships between aspects automatically? In other words: How could the Weaver determine in which order it should compose aspects that are to be woven into the same component? How could the Weaver ensure that such aspect combinations are valid?

5.3. An Aspect-Oriented Design Framework for Concurrent Systems (Constantinos Constantinides)

Presentation of the position paper submitted by Constantinos Constantinides, Atef Bader, and Tzilla Elrad

The work presented by Constantinides concentrates on the aspectual decomposition of concurrent object-oriented systems. In his presentation, Constantinides categorised aspects as intra-method, intra-object and intra-package according to their hierarchical level of cross-cutting. He also identified certain restrictions imposed by current technologies that use automatic weavers and domain-specific as well as general-purpose aspect languages.

He proposed a design framework where the overall behaviour is made up of the functional behaviour, the aspects of concern and a moderator class that coordinates the interaction between aspects and components while observing the overall semantics.

As aspects can cut across components at every level, the moderator is considered a recurring pattern from intra-method to intra-package. The design framework provides an adaptable model and a component hierarchy using a design pattern, and it allows for an open language where new aspects (specifications) can be added and their semantics can be delivered to the compiler through the moderator. The moderator is a program that extends the language itself.

The framework maintains an aspect bank which is a 2-dimensional composition of the system in terms of aspects and components that can be used to verify the inter-relationships of the aspects and which the moderator will initially consult in order to collect the required aspects.

The goal of the work presented is to achieve separation of concerns and retain this separation without having to produce an intermingled source code. The presenter and his co-authors view weaving as a general mechanism through which one can achieve composition of concerns. As such, the framework performs weaving at compile-time and the intermingled code exists only at the binary level.

Questions and Discussion

Connected to this talk a discussion arose about the use of a manual design pattern approach (as in the presented work) versus automatic weaving.

5.4. Runtime Implementation of Aspects using Kava (Ian Welch)

Presentation of the position paper submitted by Ian Welch and Robert J. Stroud

Kava allows runtime implementation of aspects where aspects are represented by meta-objects. Kava implements runtime behavioural reflection by inserting meta-level interceptions into compiled Java classes. The scope of the runtime meta-object protocol is determined by the particular choice of meta-level interceptions implemented at load-time. The meta-level interceptions switch computation from the base to the meta-level. The meta-level is composed of co-operating meta-objects. Each meta-object can be thought of as an aspect.

Kava falls between the poles of static and dynamic weaving. With static weaving aspects are implemented directly into the base level. This increases the efficiency of the system (no inefficient re-directions) and makes it easier to validate the composed system. However it reduces the runtime adaptability of the system. This can be achieved with dynamic weaving where aspects are kept separate and their binding to the base level can be adjusted at runtime. However, reduces efficiency and makes validation difficult. Kava manages to retain some of the benefits of static weaving (efficiency and easier validation) while retaining some of the benefits of dynamic weaving (adaptability). This is because the meta-level interceptions are static and the meta-level is dynamic.

The separation between meta-level interceptions and the meta-layer eases some of the problems of debugging intertwined aspect and base code. As the aspects are implemented using standalone Java classes these can be developed and debugged before being combined with the base level. What remains an open issue is ensuring that the result of combining aspects and base level results in the intended overall behaviour. This is a problem for formal validation as much as a debugging problem.

The current version of Kava allows interception and redefinition of how methods are received, methods are sent, access to fields and interception of initialisation and finalisation. The authors are working on extending Kava to allow self and non-self method invocations to be distinguished, to support inheritance of meta-objects/aspects and interception of exception raising. More information in: `http://www.cs.ncl.ac.uk/people/i.s.welch/home.formal/kava.`

Questions and Discussion

Q: [Cristina Lopes] Why not do weaving at load-time as well?
A: Because we also want to do dynamic weaving
!: [Gregor Kiczales/Jane Pryor] Inheriting aspects is difficult and not yet well understood. This appears to be an open issue in general: how to reuse and extend aspects (inside the aspect domain).

Q: [Jane Pryor] why distinguishing between base-level and aspect designers
!: [Cristina Lopes] The moment of weaving seems to depend on the -nature of- the target language. This is reflected by the presentations that addressed different languages such as Smalltalk and Java.
!: [general agreement] The term 'weaving' refers to a general process, which is not (necessarily) tied to e.g. code (pre-)processing, but to the collection of activities that together cause the effective merging of aspect and base level code.

6. Aspect Language Designs

This session was chaired by Crista Lopes. The four presentations in this session dealt with various issues in the definition of aspect languages: respectively about specification of join points, abstractions to improve aspect specifications, aspect-oriented higher-order functional programming, formalisation of aspects (and composition in a more general sense), and finally a presentation about the link between AOP and the existing concept of super-imposition.

6.1. Aspect-Oriented Programming in BETA Using the Fragment System (Jørgen Knudsen)

Presentation of the position paper submitted by Jørgen L. Knudsen

The presentation started out by giving a brief overview of the BETA programming language. The language is a compiled, strong statically typed, object-oriented language, designed to support industrial strengths system development with emphasis on software engineering principles. The first public implementation dates back to approximately 1988.

The talk then turned the focus to the support of aspect-oriented programming in the BETA language as supported by the Mjølner System. The cornerstone to this support is the Fragment System, which is designed to support issues like separate compilation, separation of concerns, information hiding, inter-module dependencies, etc.

The core of the talk was a presentation of several examples of aspects as programmed in BETA using the Fragment System. These examples illustrated the basic concepts of the Fragment System, such as fragments, fragment groups, slot, origin and dependency graph. The principle is, that source code is written in the form of fragments (syntactically legal pieces of BETA code). Fragments that are somehow related are placed in fragment groups (typically files on the file system). In source code, parts may be left unspecified by inserting so-called slots. A slot is the specification of a join point for source code to be specified elsewhere and in the future. Slots and fragments are named and typed (by the syntactic category of the source code). Source code (in the form of fragments) is connected with slots through the origin specification in the fragment groups (origin specifies the name of another fragment group). That is, a fragment is

bound to a slot if (and only if) a corresponding slot is found either in the same fragment group, in the fragment group referred to by origin, or in the origin of the origin, etc. The origin specification is one of the ways to construct the dependency graph.

After having presented how to augment classes with new aspects (such as adding a colour aspect to an entire pre-existing class hierarchy of graphical objects, the talk discussed issues like the support for inter-dependent aspects in BETA, and how the Fragment System controls the visibility of aspects.

The final part of the talk was a presentation of the approach to statement aspect weaving. Statement aspect weaving is done using exactly the same mechanisms as described above, namely fragments and slots. This implies that statement join points are specified in the form of slots in the imperative part of the program, implying that statement join points must be anticipated. This allows insertion of statements at any point in the source code, and static specification of statement join points. This approach follows the general design philosophy of aspects in BETA, namely that the location of join points is a deliberate design decision to be made by the designers (following the general principles of predictability of systems, even in the case of extensible systems).

The talk concluded by presenting the workings of the Fragment System Weaver by illustrating how the source code of different aspects are woven into the program source code to give the entire source code of the resulting executable.

Questions and Discussion

Q: [Yossi Gil] Can you use the same fragment in several places?
A: No. (this effectively means that crosscutting of fragments/aspects is not possible)
Q: [Yossi Gil] So it is the same as literate programming?
A: The fragment system uses the same weaving structure, but it is strongly typed.
Q: [Liz Kendall] Can you give examples that illustrate its industrial strength?
A: Apart from the Mjølner System itself, there are no large industrial applications.

6.2. On the Role of Method Families for Aspect-Oriented Programming (Katharina Mehner)

Presentation of the position paper submitted by Katharina Mehner and Annika Wagner

Katharina Mehner presented in her talk a language extension that aims at improving the reusable definition of aspects. The starting point was the observation that the building blocks of current aspect languages such as AspectJ are not properly encapsulated and thus cannot be referred to by a name. Aiming at the reusable definition of aspects, the possibility of factoring out common properties

within an aspect and between aspects assigned to classes in an inheritance hierarchy is highly desirable. But it is impossible without a proper encapsulation of their building blocks.

On the other hand, it was proposed in the talk that a solution for making aspect definitions reusable has to fit with the connection technique, i.e. the join points between classes and aspects. Since different methods either of the same class or from super and sub classes show the same behaviour seen under the perspective of an aspect, the building blocks of aspects are defined on a per method basis, either for one method or for a list of methods. Hence methods form the join points.

These requirements are ideally fulfilled with the concept of *method families*, i.e. equivalence classes of methods. A method family consists of its name, a method set and of attached behaviour. For the purpose of reuse method families can be referred to by their name. Thus, a kind of encapsulation of and interface to the building blocks of aspects is achieved, similar to the functional units of classes, the methods. Moreover, the usual set theoretic operations allow for a powerful, semi-automatic extension and restriction mechanism of method families during reuse. It was shown that the concept of method families is applicable to general aspect oriented languages as well as to domain specific languages.

Lessons learned: The main conclusion was, that method families are a useful extension. Moreover, they are a contribution to the question how inheritance or refinement can be carried out for aspects. While at the moment method families are completely part of the aspect language, a future step brought to discussion during the workshop was to make the assignment of sets to the building blocks of aspects part of a connector language between the class code and the aspect code. However, in the opinion of the speaker this cannot lead to more freedom for the connectors as they still have to obey the same concepts for inheritance as before.

6.3. Adaptation of Functional Object Programs (Ralf Lämmel)

Presentation of the position paper submitted by Ralf Lämmel, Günter Riedewald, and Wolfgang Lohmann

The talk was concerned with aspect-oriented higher-order functional programming. Certain operators for program transformation suitable to model aspects were suggested. The program examples were based on functional program modules (i.e. components) implementing language interpreter fragments. Environment propagation, error handling and the introduction of the monadic style were considered as aspects. The approach is based on previous work of Ralf Lämmel, Günter Riedewald et al. on meta-programming and aspect-oriented programming for first-order declarative languages.

In the 1997 ECOOP-AOP workshop Wolfgang De Meuter suggested monads as a contribution to the theoretical foundation for AOP. Ralf Lämmel's talk discussed several limitations and drawbacks of the monadic approach, e.g. the use

of a single composed monad versus different conceptual layers of effects, or tangling arising from the monadic style. One important conclusion of the talk was that a more general instance of aspect-oriented programming can be obtained based on a transformation-oriented approach. Monads still provide an important tool in such a setting. The component programmer is no longer required to code in the monadic style, but the monadic style is regarded as a kind of aspect which can be incorporated in the component code by a corresponding transformation. It is clear that such a delayed installation of the monadic style improves flexibility because the installation can pay attention to the actual context, that is to say different layers of effects can be supported and the functions which need to be computed in the different layers can be selected accordingly without overspecification.

The talk mentioned different frameworks to formalise the program transformations, e.g. term rewriting, natural semantics and functional meta-programs. It was supposed that the transformation operators operate on entire functional program modules (i.e. components). This is in contrast to Fradet's and Südholt's approach presented at the 1998 ECOOP-AOP workshop, were an additional weaver applies given transformations until a fixpoint is reached. In other technical papers Ralf Lämmel et al. developed suitable preservation properties for program transformations to facilitate formal reasoning.

The talk concluded with an overview on a kind of operator suite for program transformation that is under development. The operators are meant to model the basic roles in aspect code. There are for example operators addressing data flow issues, the insertion of computations, the installation of sum domains etc. In this sense, the talk also emphasised that program transformations provide a sensible tool even in adapting higher-order functional programming. In contrast, most previous work on program transformations for functional programs was concerned with optimisation issues.

Besides the operator suite the most interesting problem for further investigation is the question how the transformation-based approach can be applied for the common component languages rather than declarative languages.

6.4. Formalising Composition Oriented Programming (Mark Skipper)

Presentation of the position paper submitted by Mark Skipper and Sophia Drossopoulou

Mark presented a snapshot of work in progress to make a formal model of the kind of object-oriented composition mechanisms that underpin AOP and SOP. He motivated the work by pointing out that there exists no way to give an abstract description of the semantics of composition used in current AOP implementations.

He introduced the language Ku: a small imperative OO language with classes methods and attributes. The definition of Ku includes type rules, an operational semantics and a soundness property which cross-checks these against one another. It also includes a composition function that combines collections of classes

known as units. Composition attempts to merge classes methods and attributes if they correspond. Merging is undefined for corresponding entities if they do not also match.

Currently correspondence is defined as name equivalence for classes, attributes and methods; matching is defined as signature equivalence for attributes and methods; classes match if all their corresponding attributes and methods also match. This framework allows various hypotheses concerning composition to be formulated and tested. One such hypothesis: that composition preserves type correctness of composed units, was presented and discussed in more detail.

Different kinds of composition can be investigated in the framework by, for example, changing the definitions of correspondence and matching. Mark concluded his presentation with a description of the many directions in which this work can be taken in the future.

6.5. Aspects and Superimpositions (Joseph Gil)

Presentation of the position paper submitted by Shmuel Katz and Joseph Gil

The purpose of this presentation was to draw the attention of the AOP community to an extensive line of research, which started at the eighties, on the topic of superimposition. Joseph (Yossi) Gil, the presenter, pointed out the many similarities between AOP and superimposition, which are both mechanisms for separation of concerns in program design, orthogonal to the usual breakdown into modules. Traditionally, the research on superimposition was motivated and focused on the design of distributed algorithms. Properties such as liveliness, robustness to failures, deadlock prevention, etc., are best dealt with as a superimposition of a generic algorithm that insures them on the main distributed computation algorithm. In contrast, AOP is more general purpose and does not focus in any specific application domain. AOP can however, can, and should be tested and demonstrated against the many examples used in superimposition. On the other hand, there should be work on extending superimposition to other application areas.

Yossi Gil called for researchers working on AOP to build upon some of achievements of the work on superimposition. These include the syntax and language design, as proposed in Katz's 1993 TOPLAS paper which address the problem of join points, the taxonomy of superimposers.

More importantly, superimposition enjoyed the blessings of formal methods, and in particular program verification. There was research to show that the application of a superimposer to any base algorithm is correct, provided that the base algorithm satisfies certain conditions. Just as early work on superimpositions took a macro-like code implementation view of their meaning, aspects have been described in this way. However, it was found more effective to view superimpositions as separate entities, with various ways of combining them with basic systems and with each other, using a binding operator. Treating aspects in this way will also provide a firmer theoretical basis and clearer semantics for

this important modularity concept. He stressed that there is a great potential for a similar modular specification and verification techniques in AOP, however, for this to happen, the notion of generic aspects, i.e., aspects that can take parameters must be properly introduced. A group lead by Prof. Katz is currently doing research at the Technion along some of these lines.

Questions and Discussion

Q: [Lodewijk Bergmans]: What are the lessons learned by the superimposition community?
A: At least three lessons have been learned: (a) you can prove properties (b) how to apply parameterised aspects (c) many working examples exist and can be studied.
!: [Mario Südholt] Aspects should be declarative, but proving properties requires a lot of specification and proof effort
!: [Cristina Lopes] We can learn a lot about AOP by looking into work that has been done in the past and that didn't have the word "aspect" in it explicitly.

7. Wrap-Up

Moderator: Gregor Kiczales

The purpose of this session was to revisit key issues from the day, while making sure everyone contributed to the session. The set-up was the following:

> "Imagine N years into the future, when aspects are in widespread use. The dominant aspect language is called Port, and it builds on objects. There are several Port Development Kits (PDKs) and design tools. What properties do Port and PDKs have?"

The discussion was driven by a fixed list of issues that were addressed in the previous sessions (but not all issues made it to this last session!). For each issue, there were quick arguments pro/against, with the constraint that each participant could speak in at most two issues of the list. So participants had to chose their interventions wisely. At the end of each brief discussion there was a vote, to have an idea of the most popular properties of Port. What follows is the summary of the issues that were discussed.

Aspects in the Software Lifecycle

There was a consensus that there would be explicit support for aspects in all phases of the software lifecycle.

Specification Systems

There was a consensus that specification systems with explicit support for aspects should be language independent.

As for the composition of aspect specifications, it was argued that automatic composition (vs. manual composition) would be desirable, but that it was a hard problem. The majority of participants agreed that it would be sufficiently good to reach half-way between automatic and manual composition of aspect specifications.

Re-engineering Support for Aspects

Would there be tools for crosscutting binary COTS software?

The position paper by Welch and Stroud suggested so, by describing one particular technique for weaving aspects at load-time. Other possibilities include using the APIs or using advisory tags. The contra-argument was that aspects should apply only to source code, because they make serious assumptions about the implementation.

At the end, 14 to 3 voted that there would be tools for supporting crosscutting of binary code.

Design Support for Aspects

Will UML of the future support aspects? The consensus was yes. The participants agreed that there needs to standard notation, standard exchange format and mechanisms for automatic code generation.

There was also a consensus that including the concept of aspect into the UML of the future would be crucial for the success of aspects.

Weaving

When will weaving occur? Design-time, coding-time, compile-time, load-time, JIT-time, run-time?

This issue generated quite a bit of discussion. The argument that got approval of the majority of participants was that this issue is not specifically related to aspects, but that it relates to many language features, and that different times are desirable for different features. Another argument was that different weaving times would be desirable or necessary for different design entities.

The votes at the end showed that the most popular weaving time would be fairly static, so at or before compile-time (total of 14 votes). But there would also be more dynamic weaving mechanisms, so at run-time, load-time and JIT-time (total of 8 votes).

Explicit Meta-programming

Several position papers gave different perspectives on this issue. For example, DeVolder, Pryor and Bastán, Lämmel et al., Knudsen, Mehner and Wagner.

The argument for programming aspects at the meta-level was that it makes the distinction clear between objects and aspects. The argument against was that meta-programming is hard.

The votes at the end reflected the division of opinions among the participants. 15 participants voted that Port would be less meta; 10 participants voted that Port would be more meta.

Means of Referring to Join Points

The space for referring to join points ranges from fixed points in the program (a la BETA fragment system) to less fixed, property-driven specifications (e.g. "the public methods of this package"). Other possibilities in between are method identifiers (e.g. "the add(Object) method of class Set"), maybe with a wildcarding mechanism, and using syntactic elements of the programs.

18 participants voted that property-based join point specifications would be more popular, and 3 participants voted that name-based or pre-identified join points would be more popular.

Wrap-Up Summary

To conclude, the following list summarises the properties of Port and its PDKs as estimated/voted upon by the workshop participants:

- Explicit support for aspects in all phases of the software lifecycle
- Specification of aspects is Port-independent; composition of aspect
- Specs is half-manual, half-automatic
- Aspects can affect binary code (important consequences for COTS)
- There is UML-ish support for aspects
- Aspect weaving is mostly static, but Port also supports more dynamic
- Weaving mechanisms that are used less often
- Join points are referred to by property-driven specifications

It should be noted that the approach that was taken to this session, collecting issues and constraining the discussion, was considered rather successful: because of the voting, everybody was involved, and because of the limitation to participate in the discussion, people would wait to make real important points, and the discussion was not dominated by a few people.

8. Participants and Position Papers

The following list shows the participants of the workshop, with their affiliation and e-mail address. If applicable, the position papers and their co-authors are given as well. Persons marked with an asterisk have presented their work during the workshop.

1. Mehmet Aksit, University of Twente, aksit@cs.utwente.nl
2. Lodewijk Bergmans, University of Twente, bergmans@cs.utwente.nl
3. Anders Bjorvand, University of Oslo, torvill@trolldata.no
4. *Lynne Blair, Lancaster University, lb@comp.lancs.ac.uk
(with G. Blair), *A tool suite to support aspect-oriented specification*
5. *Kai Böllert, IC&C GmbH, Germany, kaib@acm.org *On weaving aspects*
6. *Siobhán Clarke, Dublin City University, sclarke@compapp.dcu.ie
(with H.Ossher, W. Harrisson, P. Tarr) *Separating concerns throughout the development lifecycle*
7. *Constantinos A. Constantinides, Illinois Institute of Technology, conscon@charlie.cns.iit.edu
(with Atef Bader & Tzilla Elrad) *An aspect-oriented design framework for concurrent systems*
8. Lutz Dominick, Siemens AG, Germany, Lutz.Dominick@mchp.siemens.de, *Aspect of lifecycle control in a C++ framework*
9. Sophia Drossopoulou, Imperial College of Science, sd@doc.ic.ac.uk
(with M. Skipper) *Formalising composition-oriented programming*
10. *Yossi Gil, The Technion, yogi@cs.technion.ac.il
(with S. Katz) *Aspects and superimpositions*
11. *Maja D'Hondt, Brussels Free University, mjdhondt@vub.ac.be
(with Th. D'Hondt) *Is domain knowledge an aspect?*
12. Theo D'Hondt, Brussels Free University, tjdhondt@vub.ac.be
(with M. D'Hondt) *Is domain knowledge an aspect?*
13. *Elizabeth A. Kendall, Royal Melbourne Institute of Technology, kendall@rmit.edu.au, *Aspect-oriented programming for role models*
14. Gregor Kiczales, Xerox PARC, gregor@parc.xerox.com
15. *Jørgen Lindskov Knudsen, University of Aarhus, jlk@daimi.au.dk
Aspect-oriented programming in BETA using the fragment system
16. *Ralf Lämmel, University of Rostock, rlaemmel@informatik.uni-rostock.de
(with G. Riedewald & W. Lohmann) *Adaptation of functional object programs*
17. *John Lamping, Xerox PARC, lamping@parc.xerox.com
The role of base in aspect-oriented programming
18. Cristina Lopes, Xerox PARC, lopes@parc.xerox.com
19. *Katharina Mehner, University of Paderborn, mehner@uni-paderborn.de
(with A. Wagner) *On the role of method families in aspect-oriented programming*
20. *Luca Pazzi, University of Modena, pazzi@unimo.it, Explicit aspect composition by part-whole state charts
21. *Jane Pryor, UNICEN, Argentina, jpryor@exa.unicen.edu.ar
(with N. Bastán) *A reflective architecture for the support of AOP in Smalltalk*

22. *Lionel Seinturier, University Paris 6, Lionel.Seinturier@lip6.fr
JST: an object synchronisation aspect for Java
23. *Mark Skipper, Imperial College of Science, mcs@bcs.org.uk
(with S. Drossopoulou) *Formalising composition-oriented programming*
24. *Mario Südholt, Ecole des Mines de Nantes, Mario.Sudholt@emn.fr
(with P. Fradet) *An aspect language for robust programming*
25. *Junichi Suzuki, Keio University, suzuki@yy.cs.keio.ac.jp
(with Y. Yamamoto) *Extending UML with aspects: aspect support in the design phase*
26. *Simon Thompson, BT Labs, Simon.2.Thompson@bt.com
(with B. Odgers) *Aspect-oriented process engineering*
27. *Ian Welch, University of Newcastle upon Tyne, i.s.welch@ncl.ac.uk
(with R. Stroud) *Load-time application of aspects to Java COTS software*
28. Edward D. Willink, Racal Research Limited, UK, Ed.Willink@rrl.co.uk
(with V. Muchnick) *Weaving a way past the C++ One Definition Rule*

References

[Bougé 88] L. Bougé and N. Francez. *A compositional approach to superimposition.* In ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pages 240–249, Jan 1988.
[Harel 77] D. Harel. *Statecharts: A visual formalism for complex systems. Science of Computer Programming*, **8**:231–274, 1987.
[Katz 93] S. Katz. *A superimposition control construct for distributed systems.*, ACM Trans. on Programming Languages and Systems, 15:337–356, April 1993.
[Pazzi 97] L. Pazzi. *Extending StateCharts for representing parts and wholes.*, In Proceedings of the EuroMicro-97 Conference, Budapest, 1997.