Advanced Separation of Concerns

Johan Brichau[1], Maurice Glandrup[2], Siobhan Clarke[3], and Lodewijk Bergmans[4]

[1] Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussel, Belgium
johan.brichau@vub.ac.be
[2] University of Twente, Dept. of Computer Science/Software Engineering, PO Box 217, 7500 AE Enschede, Netherlands
glandrup@cs.utwente.nl
[3] Department of Computer Science, Trinity College, Dublin 2, Ireland
Siobhan.Clarke@cs.tcd.ie
[4] University of Twente, Dept. of Computer Science/Software Engineering, PO Box 217, 7500 AE Enschede, Netherlands
bergmans@cs.utwente.nl

Abstract. This document describes the results of the two-day workshop on Advanced Separation of Concerns at ECOOP 2001. The workshop combined presentations with tigthly focused work in small groups on these predefined topics: requirements and challenges for ASoC technologies, conventional solutions for ASoC problems, feature interaction, design support for ASoC and design decisions for ASoC models.

1 Introduction

Recent approaches such as adaptive programming, aspect-oriented programming, composition filters, hyperspaces, role-modelling, subject-oriented programming and many others, have enhanced object-oriented programming by providing separation of concerns along additional dimensions, beyond “objects”. A series of related workshops on “Composability in OO”, “Aspect-Oriented Programming” and “Aspects & Dimensions of Concerns” that have been held at ECOOP, as well as related workshops at ICSE and OOPSLA, indicate a fast growing interest in this area. This year another workshop on this topic was organised, entitled “Advanced Separation of Concerns” (ASoC).

During ECOOP 2000 a predecessor of this workshop was held. In that workshop, two days were spent mostly on group work, delivering concrete joint results [30]. This year we decided to continue this successful set-up: the workshop combined tightly focused work in small groups with regular short plenary sessions. The following agenda shows how group work and presentations were interwoven, with plenary discussions at the end of each day:

Day 1
Introduction, division in groups, other organisational stuff
Invited talk: *Harold Ossher*
Group work
Invited talk: *Awais Rashid*
Group work

Invited talk: *Olivier Motelet*
Group discussion: "Strategic Issues for the ASoC Community"
(*Gregor Kiczales*)

Day 2
Invited talk: *Kris De Volder*
Group work
Invited talk: *Klaus Osterman*
Group work
Invited talk: *Bart De Win*
Wrap up: all groups present their findings
Panel discussion "What have we learned, where should we go?"
(chair: *Mira Mezini*)
Collaboration with the Workshop on Feature Interaction

The group work took place in 6 different focus groups, with 5 to 6 people each. The topics for the focus groups had been determined before the workshop. The following list summarises all focus groups:

1. Challenge problems for ASoC models
2. Requirements for ASoC models
3. Conventional solutions and counter examples
4. Characteristics and design decisions of ASoC models
5. Design support for applying ASoC
6. Feature Interaction for ASoC models

At the end of the first day, there was a session, led by Gregor Kiczales, on strategic issues for the ASoC community. Most of the session was dedicated to his reflection upon the current and future position of the ASoC community: he noted that the ideas are spreading with tremendous speed, and many people from both research and practice are interested in, experimenting with and/or working on ASoC technology. Examples of the growing interest are articles on ASoC (AOP) that appear in a broad range of publications, and the first conference on Aspect-Oriented Software Development, to be held at the University of Twente, The Netherlands, on 23rd to 26th of April, 2001. However, he warned that this success could backfire, because the research community has almost no time to develop and test the ideas and techniques rigorously. In particular, there are essentially no real industrial applications of ASoC techniques that can proof the applicability in practice.

The second day was concluded be a number of plenary sessions: first a wrap-up where all the focus groups presented their findings. The reports of the focus groups can be found in Sect. 3 of this paper. This was followed by a brief panel discussion, intended to raise controversial issues in AOSD. The panellists were representatives of each of the focus groups.

The panel mainly discussed the following two subjects:

– Are overlapping concerns equal to crosscutting concerns? (or: what is the definition of 'crosscutting'): An example was given where two concerns, implemented using

ASoC techniques, only needed to be composed together to form the system. Hence, they were not really crosscutting, they simply overlapped. Now, is an overlapping concern really crosscutting or not? The main response was that a crosscutting concern should at least ’partially overlap’ with ’possibly many’ other concerns. Hence, overlapping concerns would be a specific case of crosscutting concerns where they only crosscut one single other concern.

– Explicit hooks (calls) versus implicit places (obliviousness): Can you achieve separation of concerns without ’obliviousness’? The main response here was that ASoC techniques require ’oblivousness’ because otherwise full separation between the concerns cannot be achieved.

The final agenda item was a joint session with the Feature Interaction workshop. This session was inspired by the fact that the problem of feature interaction is particularly relevant to the ASoC community. This is mainly because new forms of composition introduce new composability problems, and particularly because there is a strong relation between crosscutting aspects and features. For the contents of this session we refer to the report of the focus group on “Feature Interaction for ASoC Models” in Sect. 3.2 and the workshop report of the feature interaction workshop (elsewhere in this volume).

This workshop report consists of summaries of the presentations in Sect. 2, and reports for each of the focus groups in Sect. 3. An overview of the position papers and their authors is shown at the end of this report. The position papers are available at the workshop website:
(`http://trese.cs.utwente.nl/Workshops/ecoop01asoc/`).

2 Presentations

This section contains summaries of each of the invited presentations at the workshop. For further details we refer to the position papers that were submitted by the presenters.

2.1 Some Micro-Reuse Challenges

Presenter: Harold Ossher

One of the advantages of advanced separation of concerns approaches is that the modules they provide are often better units of reuse than traditional modules. This is because what one often wants to reuse is a *collaboration*, which involves multiple classes or objects, but only fragments of each. Such collaborations can be encapsulated by modules in ASoC technologies, suggesting that their reuse will be eased. However, to realise reuse, it is necessary to be able to compose these modules in multiple different ways in different contexts, often multiple times within the same program. This presents some new challenges to the composition-mechanisms of such technologies.

Consider the implementation of a ‘Link’ feature, which implements the ability to chain objects together using links stored within the objects. Secondly, it also counts the number of times its next() method is called.

```
public class Link {
    private Link _link;
    private int _count=0;
    public link next() {count++; return _link;}
    public void setNext(Link l) { _link = l;}
}
```

Clearly, `Link` is a highly reusable abstraction. Composing it with another class adds a link capability to that class (e.g. using Hyper/J). But within a single system, one might want to add links to many different classes or even to the same class (e.g. if a class should be doubly linked). In many cases, different problems arise:

1. What if Link is composed with a class `D` that already has a `_link` instance variable? Do we want them to be shared or should they be different? In most cases this will be the latter, but we could think of situations were the opposite is wanted. In such a case, an even more difficult issue arises: The class `D` may well contain accesses to its `_link` variable. But the `Link` class' access counter `_count` depends on the protocol that all access to the `_link` variable occur through the `next()` method. Thus, accesses in class `D` to `_link` will not be counted.
2. If `Link` is composed twice or more with the same class, do we want them to share their `_link` and `_count` instance variables or not? Clearly, in case of a doubly linked chain of objects, we do not want `_link` to be shared, but we might want to share `_count`.

It becomes even more difficult if `_count` is static (one `_count` per class):

```
    private static _count=0;
```

1. If Link is composed with a class and a subclass of this class, do we want separate _count variables for the class and the subclass or should _count be shared by the class and its subclass?
2. Or should _count be unique for the entire system? This amounts to interpreting the “static” relative to the Link class, rather than the classes with which it is composed.

Many subtle and important issues arise when one is trying to use composition or weaving to achieve reuse. As soon as the same fragment is composed in different ways into different contexts, issues arise how to specify exactly the desired semantics, especially of sharing. Many possible alternatives exist and we should allow any of them to be expressed.

2.2 Sophisticated Crosscuts for E-Commerce

Presenter: Olivier Motelet

Expressing crosscutting modularity of a certain concern involves the definition of where (or when) a certain action should be performed. ASoC technologies should support sufficient expressive power to separate this definition of the crosscut from the action definitions. Inadequate expressive power for crosscut definitions will most often lead to action definitions containing part of the crosscut definition. This is illustrated in the context of a simple example:

Consider an e-commerce application running a shop's website where clients can buy products online using a web browser. A concrete usage scenario can be as follows: The client searches for a specific product and orders it, then performs another search that he cancels and finally he purchases the result of a third search. This scenario can be expressed using the following sequence of events:

```
search; buy; search; back; search; buy
```

The given scenario illustrates the base functionality of the application. Additional behaviour, like a discount and security policy, are crosscutting the base functionality and are most preferably handled using ASoC technologies. In the example, the ASoC technology that is used to implement them is event-monitor based. This means that the base program generates events during its execution (such as those shown in the example above). A crosscut is defined as a pattern of events and hence, and aspect is defined as follows:

```
aspect = when aPatternOfEvents perform action
```

The event-monitor traces the events and executes the `action` when `aPatternOfEvents` occurred. First, we present the implementation (in a Java-like syntax) of the discount policy using a simple crosscut mechanism:

```
aspect Discount {
        boolean firstbuy = true;
        float discountRate = 1.0;

        when buy perform {
          if (firstbuy)
            firstbuy = false;
          else
            discountRate -= 0.01;
            price *= discountRate; }
        }
```

This aspect is executed each time a user buys a product. It applies a discountRate to the price of the product after the second purchase of the user. However, the action-code (after the perform keyword) of this aspect still contains book-keeping code to check if this buy-event is not the first buy-event. In other words, the action code still contains code to define a sophisticated crosscut "all buys, except the first". A more sophisticated crosscut definition mechanism should separate the definition of the crosscut and the action. This is illustrated using the same Discount aspect example:

```
aspect Discount {
        float discountRate = 1.0;
```

```
when enableDiscount() perform {
  discountRate -= 0.01;
  price *= discountRate; }

Crosscut enableDiscount() {
  Event e = nextEvent(buy);
  return enableDiscount2(); }

Crosscut enableDiscount2() {
  Event e = nextEvent(buy);
  {return new Crosscut(e);
    |||
  return enableDiscount2()}
}
```

Here, the action code is no longer tangled with book-keeping code. The pattern of events is implemented by the function enableDiscount(). This function skips the first occurrence of the buy-event and calls the enableDiscount2() function. The latter returns a crosscut when a buy-event occurs and does a recursive call in parallel. This ensures that the action is executed and that future buy-events are also captured.

This separation makes the aspects easier to understand: the programmer can read crosscuts and actions separately. The aspect specifications are also more reusable since actions and crosscuts can be modified separately.

Position Statement: ASoC tools should provide a sufficiently expressive crosscut mechanism that supports the definition of sophisticated crosscuts, allowing a clear separation between the crosscut-code and the action-code of an aspect.

2.3 Code Reuse, an Essential Concern in the Design of Aspect Languages?

Presenter: Kris De Volder

Code-scattering, that results from cross-cutting concerns, includes *code replication* and *code tangling*. Code replication means that the same code is repeated in different locations, while code tangling means that code for different concerns is located in the same location. To implement crosscutting concerns in a satisfactory way, an ASoC technique needs to address both these issues. Code-tangling can be addressed by crosscutting modularization mechanisms, whereas code-replication can be addressed by mechanisms of genericity that make a piece of code reusable in different contexts.

Consider the implementation of a method that searches for an element in an enumerable data structure:

```
public boolean search(Object e) {
    boolean found = false;
    Enumeration elems = this.elements();
    while (!found && (elems.hasMoreElements()))
        found = e.equals(elems.nextElement());
    return found; }
```

This method should be located in many different classes (e.g. all classes that understand the *elements()* message). Since these classes are not located in the same class-hierarchy, it clearly is a crosscutting concern and it should be modularised using a crosscutting modularisation technique. This simple example illustrates both the issues of separation (the code is scattered across different classes) and replication (the scattered code is similar or, in this case, even identical).

A Logic Meta-Programming (LMP) approach can support both separation and replication in a natural way:

Separation: The representation of a base program as a set of logic facts allows for naturally crosscutting modularity structure because the facts can be grouped into logic modules without being constrained by the modularity structure of the program they describe. In the example this means that we will have a logic module consisting of the following facts:

```
method(Stack,boolean,search,[Object],[e],
       { boolean found=false ... return found }).
method(Array,boolean,search,[Object],[e],
       { boolean found=false ... return found }).
method(WidgetList,boolean,search,[Object],[e],
       { boolean found=false ... return found }).
...
```

Replication: Even if we can group facts together in logic modules, we will frequently end up with recurring patterns of code (as shown in the example). The generative power of logic rules and logic variables enables an expressive mechanism to deal with replication. Applying this to the example, we end up with the following rule:

```
method(?Class,boolean,search,[Object],[e],
       { boolean found=false ... return found; }) :-
    class(?Class),
     hasMethodSignature(?Class,Enumeration,elements,[]).
```

In this example, scattered code is identical in all classes. But it is significantly harder if the scattered code variates from location to location. An example of this is the accept-method in a Visitor design pattern:

```
method(?Visited,void,accept,[?AbstractVisitor],[v],
       { v.visit<?Visited>(this) }) :-

    abstractVisitor(?AbstractVisitor),
    visitorVisits(?AbstractVisitor,?Visited),
    concrete(?Visited).
```

There are three variation-points in the above example: the name of the receiver class (?Visited), the type of the visitor being accepted (?AbstractVisitor) and the name of the visit-method (visit<?Visited>) which depends on the receiver class.

Position Statement: The issue of replication in aspect languages is as fundamental as the issue of separation. Besides a crosscutting modularization technique, a good aspect language should offer a mechanism of genericity and parameterisation.

2.4 Risk Management in Component-Based Development

Presenter: Awais Rashid

There have been many promises associated with component-based development, for example, instant productivity gains; accelerated time to market; lower development costs; simple and rapid mechanism for increasing the functionality and capability of a system; and low risk development strategy. This presentation addressed the claim of a low risk development strategy in particular. The reality of component-based development is that there are significant risks to organisations that are particularly threatening to small organisations. These risks stem from four main factors: the black-box nature of COTS software; the quality of COTS software; the lack of component interoperability standards; and the disparity of customer-vendor evolution cycles. By considering these risks as crosscutting concerns across the whole development cycle, it can be demonstrated that crosscutting concerns do not only occur within artefacts at each development stage. Higher-level, more abstract, crosscutting concerns can also be identified and managed using ASoC principles.

The risks relating to CBD can be organised in six main categories: evaluation (E), integration (I), context (C), quality (Q), evolution (Evol) and process (P) risks. Evaluation risks are associated with problems of evaluating off-the-shelf software for use in system development. Integration risks relate to the problems of composing systems from COTS software. Context risks relate to the problems of using similar components in different application contexts. Process risks are associated with the problems of using inappropriate development process. Quality risks stem from the perceived reliability of COTS components and the ease with which their capabilities can be verified. Evolution risks are related to the extended development and management of component-based applications.

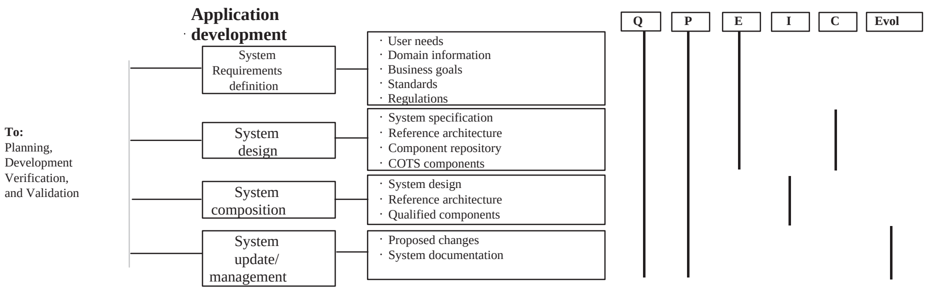

Fig. 1. Crosscutting and overlapping nature of risks in CBD

Figure 1 maps the different categories of CBD risks to development stages in a generic CBD cycle. The risks are shown to the right of the diagram. Black vertical lines indicate the development stage affected by the risk category. It should be noted that the various types of risks overlap as they affect the same development stages at times.

Separation of concerns is an approach to manage these risks. Categorising the various CBD risks into individual categories makes it possible to identify risk management mechanisms for each individual risk category (details of risk management mechanisms are in position paper 18. The individual risk management strategies may then be integrated into a single global strategy by factoring out the common mechanisms underlying the various individual strategies. Some of the core principles of advanced separation of concerns (i.e. separating crosscutting concerns in order to reason about them in isolation, with subsequent composition of the results) were demonstrated, therefore, as being useful for more than just development artefacts. The higher-level, more abstract, concern of risk management in component-based development also benefits from the approach.

2.5 Object-Oriented Composition is Tangled

Presenter: Klaus Ostermann

This presentation examined standard object-oriented composition mechanisms, such as inheritance, object composition, and delegation, and assessed difficulties associated with them within the context of advanced separation of concerns. One significant difficulty relates to the construction of mechanisms for untangling code (the standard goal for ASoC mechanisms) that are built on top of composition mechanisms that are themselves inherently tangled. A fixed set of five composition properties of standard composition mechanisms was discussed, followed by examples illustrating tangled composition scenarios in these approaches. These composition properties describe the relation that holds between two modules M and B (classes and/or objects) to be composed, where B denotes the base module, M denotes the modification module, and M(B) denotes the composition. The composition properties are:

Overriding: The ability of the modification to override methods defined in the base. In M(B), M's definitions hide B's definitions with the same name. Self invocations within B ignore redefinitions in M.

Transparent Redirection: The ability to transparently redirect B's this to denote M(B) within the composition.

Acquisition: The ability to use definitions in B as if these were local methods in M(B) (transparent forwarding of services from M to B).

Subtyping: The promise that M(B) fulfils the contract specified by B, or that M(B) can be used everywhere B is expected.

Polymorphism: The ability to (dynamically or statically) apply M to any subtype of B.

What is frequently needed is a set of composition properties which is not provided by any of the predefined mechanisms. The presentation discussed the need to clean up the primary composition mechanisms before introducing additional ASoC composition mechanisms. This requirement was underpinned through the illustration of a number of examples. It *is* possible to solve all the different composition requirements for each of the different scenarios using standard composition mechanisms such as inheritance, delegation, etc.

However, the result is deemed unsatisfactory for a number of reasons. First, depending on the desired mixture of composition properties, different architectures were used. Indeed, even where the same mixture of composition properties was required, it is possible for different designers to come up with different architectures. This problem is further compounded when a later change to the required composition features may necessitate switching to another architecture. This may require far reaching changes in the code. Even without considering the architectural issues, the design gets complex as soon as a non-standard composition is required. Together, all these problems affect the understandability, and hence the maintainability of object-oriented programs.

The basic idea that was introduced to counteract these difficulties is based upon the separation of composition properties at *language* level. With explicit linguistic means to render individual composition properties, they may be independently applicable in any architecture. This approach removes the tangling properties associated with composition mechanisms, which will therefore provide a better basis on which additional ASoC composition mechanisms may be built.

2.6 Building Frameworks in AspectJ

Presenter: Bart De Win

This presentation demonstrated the benefits that can be obtained when combining the capabilities of AspectJ with the notion of frameworks. Software reuse is an important goal in software engineering, and it was illustrated how frameworks provide mechanisms to enhance the reusability of aspects. AspectJ has mechanisms to support the coding of reusable crosscutting code with abstract aspects. Where crosscutting code requires co-operation from an application in order to work, some code is also required that is specific to a particular application (e.g., the specification of concrete pointcuts, etc.). In order to improve the reusability of aspects, a common approach is to extract the generic part of an aspect from the application specific part. One approach to deploying the reusable parts of an aspect might be to present them as a library of functions and classes. However, framework technology can improve this approach. The main advantage of frameworks over libraries is that the former can encode and enforce (to some degree) how the code should be used.

The challenge for aspect programmers within this model is therefore to design a solution in such a way that it is possible to combine a general specification with a specialised specification. First, the generic core structure should be designed using aspects. Specific concern implementations should also be provided. Then, at deployment time, any additional mechanisms may be implemented if necessary, as pointcuts specify which mechanisms to use and where. A security example, with crosscutting behaviour of access control and confidentiality, was presented to illustrate the approach. The AspectJ constructs supporting abstract pointcuts, together with aspect inheritance were shown to support the specification of reusable frameworks.

However, some difficulties of using AspectJ as a reusable framework were discussed as well. For example, when the pointcut definition is part of the framework, it is often difficult to foresee what type of parameters will be required (see Fig. 2). It is recommended that a more open mechanism than parameter passing should be considered.

Another problem that has been encountered is related to the use of pointcut definitions. Pointcut definitions are static, and may therefore only be referenced and extended in a static way (see Fig. 3).

```
pointcut checkAccessCut(int i): executions (* Resource1. foo(i)) ;
pointcut checkAccessCut(String s): executions (* Resource2.bar(s)) ;

        ⬇

abstract pointcut checkAccessCut(?) ;
```

Fig. 2. Abstract generalizations that are hard to predict.

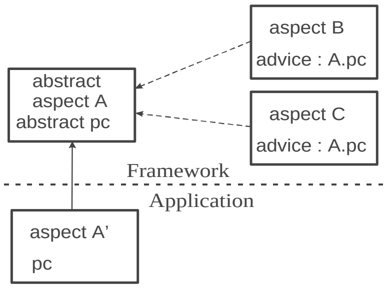

Fig. 3. Pointcut definitions are static

This makes it impossible to isolate an abstract pointcut definition into a separate construct, and rely on delegation to select one of many concrete versions, depending on where the original pointcut is used. A more open weaving process was suggested, such that pointcut definitions can be the result of a function evaluation over a representation of the application structure.

3 Focus Groups

3.1 Challenge Problems for ASoC Models

Group Members: Johan Fabry, John Zinky, Soren Top, Lahire Philippe, Marie Beurton-Aimar

Issues: Type of challenge problems, characteristics of these problems, categorise these challenge problems

The challenge problems focus group started with enumerating the issues considered important by the attendants. They are listed below:

– A first major issue was the apparent simplicity of current ASOC tools, and the clear need to go towards more advanced solutions. This would need to include support for composition of different, possibly overlapping, aspects and make it possible for the developer to get 'the big picture' of how the different aspects interact.
– A second issue is the use of existing techniques: we consider AOP, for example, to be very good at what it does: putting pieces of code in 'strange places'. However, we can not ignore the benefits of OOP, for example. The code which is weaved in should ideally contain only out-calls to code written in a 'classical' way.

– The third and final issue is the current lack of support for runtime aspects. We consider that the moment at which code gets weaved to be orthogonal to the idea of ASoC. ASoC should not be restricted only to compile-time weaving, but should also include weaving at runtime.

We envision future ASoC solutions to be more domain-driven: aspect programmes contain higher level information, domain knowledge is contained in the aspect definition, and the weaver would reason about the high-level information according to the aspect definition. Aspect definitions are now parameterise-able at a higher level: the parameters are values that are relevant for that domain. A domain expert would define a given domain as an aspect by first finding the right sets of parameters for that domain, second determine some kind of mapping from sets or ranges of parameters to an algorithm, and third define the join points. Note that this allows for a repository of algorithms that can be reused, even across different domains. The domain expert would also be responsible for determining how different aspects are composed. He can define a number of composition parameters, which allow for composition control by the programmer, where necessary. We realise that this proposed system differs strongly from current ASoC solutions. However, a first step towards our proposal would be the inclusion of libraries of aspects, which extend the current ASoC solutions.

3.2 Feature Interaction

Group Members: Johan Brichau, Guenter Kniesel, Olivier Motelet, Eddy Truyen, Mehmet Aksit

Issues: Measurement and categorisation feature interaction problems, methods to assure quality

Research in feature interaction deals with the unanticipated addition of features to a system. A feature in this context is a characteristic of the system. The focus of this research-domain is the automatic detection of conflicts between different features when a new feature is added to the system. Although feature modelling originated in the telephony domain, it is now also commonly used in software product-lines where a software product is tailored to the needs of the user who selects the desired features from a predefined set.

In the context of ASoC software development, ideally, every feature of the software system should map to one concern. Hence, in an ideal ASoC environment, the addition of a new feature to a software product means the introduction of one concern. Because ASoC technologies are improving the modularization of all concerns, the problems with feature interactions are becoming more important in ASoC research. A better separation of concerns obviously introduces more abstraction units in the implementation of a software product. Since composition always introduces interactions, the number of interactions is likely to be much higher when using ASoC compared to using more traditional techniques. What is even more important is that each feature is explicitly handled in an isolated concern and that the composition may raise issues that cannot be easily determined when the features are considered in isolation.

In order to allow detection of conflicts between features in a software product, each feature should be annotated with semantic information that defines what the feature is about. The main problem is the definition of some canonical model to specify the annotations. The problem here is two-fold:

– What should the annotations specify to allow detection of conflicts? i.e. what are the annotations about?
– What medium is most appropriate to annotate the different concerns to allow reasoning for feature interaction? i.e.: what is the language to use?

What Should Features Annotate? Features could conflict on many different levels, i.e. the conflict can be implementation-specific as well as domain-specific Therefore features should be specified at different levels of observation which can be coarse-grained or fine-grained. In the most fine-grained level we can specify every detail of the implementation (maybe even the program source code itself). More coarse-grained levels can omit the details and focus on higher level descriptions of the features (such as collaborations, architectures, use cases, domain models, etc. . .). This is illustrated in Fig. 4.

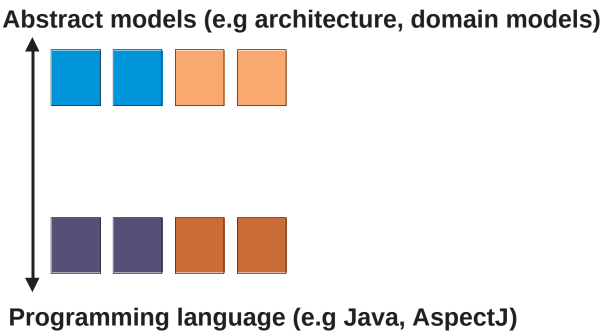
Abstract models (e.g architecture, domain models)
Programming language (e.g Java, AspectJ)

Fig. 4. Features can be defined – and conflict – at multiple levels

Consider a simple example, in which a certain class C implements a container that we extend with a 'linking' feature that implements the operations to form linked-list of containers. We also extend the same container with a 'back-linking' feature that allows backward linking in the list of containers. Obviously, operations on the list (e.g. delete, insert, . . .) that are implemented by both features will interfere. E.g., the insert operation of the linking feature only manipulates the forward-links of the list, which results in inconsistent backward-links.

A checker should detect such a 'feature interaction' problem, but the implementation itself is not sufficient. A high-level domain description is necessary, which in this case could be in a graph-language.

What Language to Use? Adding a new feature to a system means transforming a system. Features could then be described in terms of transformers. While it is generally impossible to prove that a system will work correctly after applying the transformations, we should limit the reasoning to the changes made by the transformations and check if they do not conflict in transformation. The transformations should specify their output as well as what triggers them. A possible strategy would be to say that they do not conflict if for each possible order of the application of the transformations, the end result is semantically the same.

Conclusion and Final Notes. Features of a software system will always interact and interfere due to issues that could not be observed when the features were considered in isolation. These issues can be implementation-specific as well as domain-specific. Therefore, features should annotate their impact on different levels of detail to allow conflict-detection.

Given this reasoning framework, there are still a lot of open research questions:

– Which category of features may interfere?
– What are the desired specification levels?
– How do we define an interference specification language in a particular (generic) domain?
– How to ensure conformance and traceability of refinement among the levels?
– How do we implement the checker algorithm in a practical way?
– How to implement the 'Feature interaction resolution' as a separate concern, once a certain conflict has been detected?

3.3 Design Support for Applying ASoC

Group Members: Maurice Glandrup, Siobhán Clarke, Constantinos Constantinides, Awais Rashid, Amparo Navasa Martinez

Issues: Type of requirements for ASoC models; examples of requirements for ASoC models; technologies to support ASoC requirements

The design support for applying ASoC focussed on the following type of questions:

– How should the different kinds of concerns be modelled at different stages of the software life cycle and in particular at 'design time'.
– Identify ASoC issues in the specification, analysis and design phases of the software development process.

The aim was to get a set of scenarios that highlight requirements on ASoC support across the life cycle; a set of issues and trade-offs to be addressed in providing such support; requirements on solutions. The heart of the problem in providing design support for ASoC can be summarised as follows:

Aspect (or concern)-orientation can be seen as a new direction in software development that requires its own process, modelling techniques and tooling environment. This is, in a way, similar to object-orientation that also went through this process to set-up an environment to develop quality software. Recognising aspect/concerns in problems and modelling the solution with help of the recognised concerns, is largely an unknown area. There are very little design heuristics to develop applications in a concern-oriented manner. This also accounts for process and modelling techniques.

To illustrate one view on the problems, we use an example: One of the issues to consider in aspect-oriented modelling and designing are the concerns that different stakeholders involved in the development of the software have, and how these concerns crosscut. For example, suppose in an imaginary project the following stakeholders with their concerns (in the remainder of the example we focus on the Cost concern):

Stakeholder	**Viewpoint (concerns of stakeholder ordered by importance)**			
Marketer	Cost	Functionality	Evolution	
Customer	Functionality	Technology	Cost	Cost
Architect/ designer	Evolution	Technology	Functionality	Cost
Programmer/ developer	Performance	Technology		

The domain knowledge of every stakeholder gives the stakeholder a certain viewpoint on a concern. For instance, the marketer wants low cost for a certain functionality to make a good profit, whereas the customer wants functionality he can use that is realised in a non-proprietary technology with low cost. Architects and programmers do not have cost as their primary concern, however, they realise that cost drives their development.

Certain design decisions are guided through cost. For example, design decisions that make the design more flexible, take often more time to develop than straight forward design constructs that are less flexible. What design decision to take depends on, for instance, the cost concern. Cost does influence design decisions, but it is hard to make cost explicit in the design of software. When giving design support for applying ASoC it is important to know how cost is “valued” in the project.

Analysing the above, we can say that different stakeholders recognise the same or similar concerns, but because of their domain knowledge and their viewpoint, concerns are valued differently. Concerns can crosscut stakeholder viewpoints; not in terms of meaning, but in terms of value in the project. The value of a concern expresses its dominance for a stakeholder.

Concerns should be modelled using the most suitable technique. However, translating the value of a concern in such a way that it is comparable with other concerns is a problem. In our example, the architect values the evolution concern very high, while cost is a lesser concern, the marketer will do vice versa. Modelling evolution is fundamentally different from modelling cost. Because of this difference, comparing and valuing evolution and cost is difficult. A number of issues must be considered during comparison and valuing, one of them is the value of a stakeholder to determine its decision influence for a certain concern.

To conclude the work of the discussion group, we define some research areas with possible research directions for design support for applying ASoC:

– When giving design support for applying ASoC, we need to make the concerns of the stakeholders explicit. This also accounts for the nature of the concerns; e.g. how is the concern valued, how do concerns crosscut over the stakeholders. In the example we illustrated that, it is not always possible to make concerns –that influence decision-making process during the development of software– explicit during the design of software. Valuing concerns and relating them seems therefore important in giving design support for applying ASoC. A point of research is how to value

different concerns that are modelled differently and from there give design support for applying ASoC.
- Are a number of concerns and their dominance applicable on forehand in projects? For instance, in case of the cost and evolution concern this seems to be true. A (relatively obvious) example of the dominance order could be as follows:
  - for parts of software that are technological dependent, do not invest a lot of time in designing flexible design constructs
  - for parts of the software that should be reusable and capable of evolving, do invest a lost of time in designing flexible design constructs
- A new modelling technique or direction means adjustment of existing processes or the definition of a new process. Since, at this moment, we do not know what kind of design support is necessary for applying ASoC, the best approach is to integrate the design support and its guidelines in existing processes. When there is enough experience with design support for applying ASoC, new processes can be defined, or existing ones be adjusted. For possible directions and connection points, we can use work already done in this area; such as for example the PREview system; a method for separation of concerns at requirements definition level.
- A more formal foundation for giving design support is one of the directions that were mentioned in the discussion group as a possible direction for expressing guidelines for design support for applying ASoC. The reason for formalising guidelines is the following: large complex systems will have numerous concerns and the value system how concerns are related can become very complex. Automation support can be very helpful in giving an overall view of the problems and the reasons why certain design decisions were taken. Set-theory is one of the formal theories that can be used here.

3.4 Requirements on ASoC Models

Group Members: Adeline Capouillez, Bart De Win, Eduardo Kessler Piveta, Mira Mezini, Mark Skipper, Bedir Tekinerdogan

Issues: Type of requirements for ASoC models; examples of requirements for ASoC models; technologies to support ASoC requirements

Problem Overview. In order to consider requirements on advanced separation of concerns, it is first necessary to consider why “advanced” ways to separate concerns are needed. The root of the software problem being solved by ASoC mechanisms is that many software concerns *crosscut* each other, and are therefore difficult to modularise. Two concerns, C1 and C2, are crosscutting if the following holds:

- There is a clear, modular description of both C1 and C2, in the decomposition scheme most appropriate for them.
- If we consider one of the concerns (e.g., C1) to be the reference decomposition scheme, we may be obliged to scatter at least one concept of C2 into at least one concept in C1

Object co-ordination according to the publisher-subscriber pattern and basic object decomposition are examples of crosscutting concerns. This discussion group focused on the need for flexible specification of crosscutting concerns, with their subsequent composition, as the key requirements on ASoC models.

Describing Crosscutting Concerns. This section describes some features an ASoC model should have to support the specification of a crosscutting concern. First, it should be possible to express each concern in the most appropriate decomposition scheme for it. For example, for many concerns, the object-oriented paradigm may suit its specification adequately. However, for others (e.g., our concern that handles the co-ordination of objects from before), the object-oriented paradigm does not work well. An ASoC model should not be restrictive as to the kinds of paradigms that may be used to describe any individual concern,

Secondly, the model should support the description of a concern that does not anticipate the points of interdependencies with other concerns. Of course, during composition, one concern may intervene with another. However, for the description of an individual concern, it should be possible not to discuss those intervention points. This requirement is part of a higher-level requirement on ASoC models to support the flexible specification of crosscutting concerns with *or without* details of its interdependencies. A range of possibilities is needed (Fig. 5).

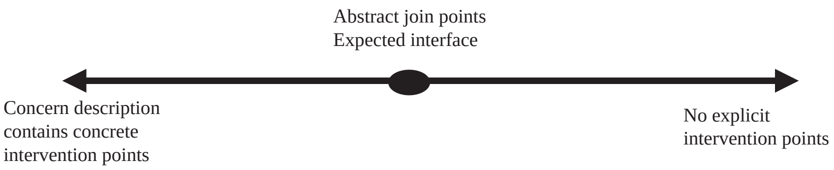
Abstract join points
Expected interface
Concern description contains concrete intervention points
No explicit intervention points

Fig. 5. Range of valid concern descriptions

Finally, concern providers should be able to declare concern properties relevant for future deployments of the system. However, these properties should not include interdependencies. For example, a valid property is to say that the concern has some hard, real-time constraints. However, an assertion that "authentication should be executed before access control" is not valid since it is an inter-concern property.

Of course, inter-concern properties are important. It should be possible to describe and define these properties separately from the individual concerns. These would serve as input to the composition process, together with the relevant concerns. Thus it may be possible to express the "authentication before access control" constraint from above in some abstract way that is separate from the descriptions of concerns involved.

Composition. In any ASoC model, where concerns have been separated, they must be subsequently joined in order to achieve all required tasks. This section lists requirements on the composition element of ASoC models. The following requirements have been identified:

- A powerful means to express join points is required
- Powerful adaptation mechanisms for changing the behaviour of individual concerns at join points are very important
- It should be possible to use the result of a concern composition as a concern for the next higher level composition

- Composition specification can be a first-class entity – i.e., the composition may be performed at runtime.

3.5 Characteristics and Design Decisions on ASoC Models

Group Members: Pierre Crescenzo, Krystof Czarnecki, Kris De Volder, Erik Ernst, Robert Filman, Erik Hilsdale, David Lorentz, Harold Ossher

Issues: Design decisions required for advanced separation of concerns; issues and tradeoffs required for making these decisions; interactions among design decisions; requirements on solution approaches; scenarios that highlight issues in making different design decisions.

Problem Overview. Although the complete set of issues and requirements on approaches to advanced separation of concerns is not yet known, there are a number of requirements on, and features of, ASoC approaches and mechanisms that are now considered "standard" (e.g. separation of crosscutting behaviour, composition of separately specified behaviours). There exists a design space of the features and properties of ASoC approaches. Each point of the design space is characterised by both positive and negative effects. In previous workshops, many design dimensions have been discussed for ASoC systems. This workshop focused on four of these: the interaction between genericity and aspects, ternary and binary representations, symmetry, and obliviousness.

Genericity vs. Aspects. Generics are the ability to parameterise a class (or module or procedure, etc.) with respect to elements such as types, functions and constants. Generics are seen in Ada generics, C++ templates, and the emerging Java generics package. The group concluded that some of the goals of AOP can be realised with generics, but that generics differ from AOP in that generics are an explicit parameterisation mechanism (the original programmer is parameterising the program with respect to something) while aspects support implicit, post hoc program parameterisation. The group also concluded that the relationship between generics and aspects, and how they may be integrated, is a good topic for further research.

Ternary vs. Binary Representations. The ternary vs binary issue asks the question: Where does the text (the specification) that ties the aspects to the code reside? Binary systems incorporate the specification in the code; ternary systems have a separate "file" for the specification. With binary systems, tangling the specification language with the rest of the program requires the programmer to think about it through the entire process. In order to debug the system, the programmer has to be aware of the compositions that are being done. The group also discussed possible reuse difficulties associated with tangling the composition/weaving specification with aspect code.

Symmetry. Symmetry is the issue of whether there is an identifiable base code on which aspects are applied, or if all elements are created equal, where there is some composition mechanism for building composites from elements and composites. The group concluded that the key issue is that aspect systems support closure (that is, that the "apply aspect" operation can be applied to the result of the "apply aspect" operation, and that the "apply aspect" operation can be applied to aspects).

Obliviousness. Obliviousness is the issue of the extent to which a component has knowledge of other components. An obliviousness spectrum was identified: (1) direct (named) call; (2) "Implicit invocation" [28] – it is known that something will happen, but not what. This is seen in both OO method dispatch and event mechanisms; and (3) aspects. The critical elements are what knowledge is exported. With direct, named calls and with implicit invocation, knowledge is exported explicitly by the program. With aspects, the AOP system defines what is exported. The obliviousness spectrum highlights issues associated with whether the programmer should have the ability to seal certain parts of the program from aspects (the aspect parallel to Java "private"). Possibilities (and benefits) here include giving the programmer the ability to restrict who can add aspects (for example, the development team may but customers may not). Another possibility relates to allowing the programmer to specify sections that may be "critical" – i.e., that adding some kinds of aspects to this section may be too "dangerous" (have a negative impact), but some other kinds may not negatively impact the section.

3.6 Conventional Solutions and Counter Examples

Group Members: Pascal Constanza, Lidia Fuentes, Juan Hernãndez, Gregor Kiczales, Klaus Ostermann.

Issues: Available conventional solutions to use to implement ASoC technologies, trade-offs advantages and disadvantages of conventional and ASoC approaches, the use of patterns

There were three main goals that for this focus group.

– Firstly, to identify what sort of current reuse approaches that do not use aspect-oriented techniques could be benefited from or implemented using aspect-oriented technologies. In case of the existence of such ASoC technology, the focus group tried to identify the trade-off's, advantages and disadvantages of the distinct approaches.
– And finally, to categorise which of the identified ASoC technology is best for some common scenarios/problems.

The discussion was focused on reuse technologies at different levels of abstraction because aspects and concerns can be expressed at different abstraction levels (see the summary of Feature Interaction discussion group). Consequently, the focus group considered reuse technologies not only at implementation level but also at design and architectural level.

Figure 6 summarises the conclusions regarding the goals mentioned above. Whereas the upper half of the figure depicts different aspect-oriented technologies, the bottom half shows the conventional software engineering reuse techniques, and they are explained in the following subsections.

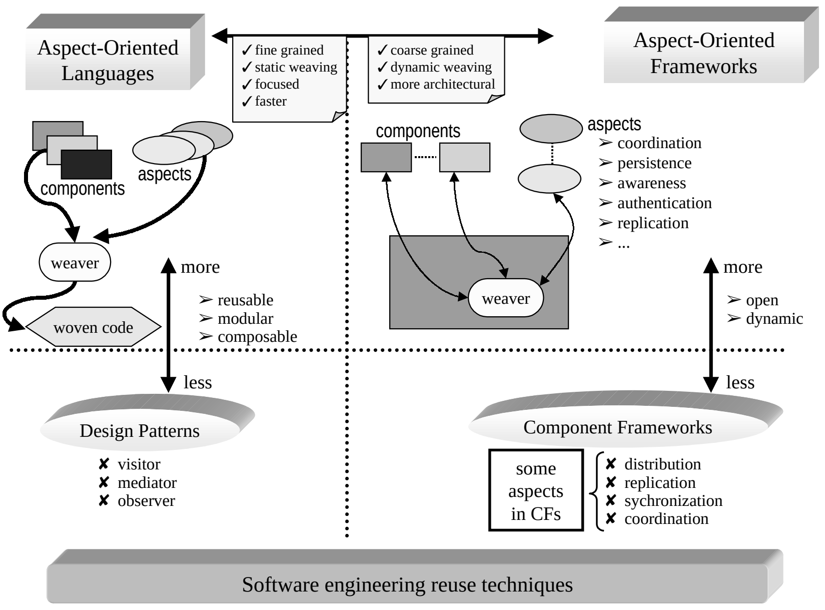

Fig. 6. Conventional solutions vs ASoC Technologies

Conventional Reuse Techniques. Frameworks and patterns facilitate reuse at different levels. Whereas the former focus on reuse of concrete designs, algorithms and their implementations in a particular language, the latter focus on reuse of abstract design and software architectures. Moreover, patterns are language independent [25].

Design Patterns. Design patterns provide a customisable solution to a concrete design problem as a set of few related classes that interact in a particular way [27]. The application of design patterns concerns the design phase of the software life cycle. Patterns may be applicable to the internal design of a component framework because they represent recurring solutions to given and known problems within a particular context. However and from the reuse point of view, design patterns show some drawbacks (position paper 16):

- First, design patterns are solutions driven by the user. Consequently, the application of design patterns to a concrete problem relies on users, so different users may produce different resulting systems.
- Besides, once a pattern is selected to provide a solution to a particular problem, it must be implemented in a given programming language. Patterns provide design reuse but not implementation code reuse. Consequently, a pattern has to be (re-)implemented every time it is applied.
- And finally, the implementation of some patterns using general purpose language like java leads to tangled code. This is the case of the observer, visitor and mediator pattern because of the invasive nature of them. Figure 7 intuitively depicts this

scenario, where the application of any of the mentioned patterns leads to code that cut across several components.

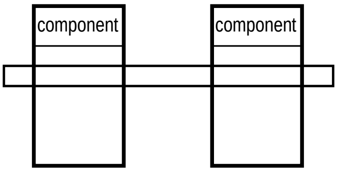

Fig. 7. Crosscutting components

Component Frameworks. During the last years, framework technology as consolidated as a suitable technology for the design and implementation of complex distributed systems. A component framework encapsulates a reusable, customisable software architecture as a collection of collaborating and extensible components [25].

Component frameworks offer a wide number of varying features, such as component interaction, distributed support and platform support among others. However, framework technology by itself is not enough to manage the complexity of middleware applications, because it does not provide the separation of concerns needed to implement the basic functionality of the components and the properties that cut across them as different entities. There are other features that are well-known aspects in the AO community that cut across functional components. This is the case of distribution, replication, synchronisation and co-ordination aspects. Aspect-oriented frameworks is a recent ASoC technology that tries to cope with these issues.

Aspect-Oriented Technologies. The previous reuse techniques may benefit from current ASoC technologies in order to solve the reusability problems mentioned in the previous section (see Fig. 6). On one hand, aspect-oriented languages such as AspectJ [29] may be used instead of general purpose programming languages for implementing patterns, producing thus components that are more reusable, modular and composable. On the other hand, the combination of aspect technologies with a framework approach leads to Aspect-Oriented Frameworks (position papers 5,16,22), which are more open and dynamic than component frameworks.

Aspect-Oriented Frameworks offer a more architectural perspective than AO languages. They are also coarse grained but they provide dynamic weaving instead of static weaving that it is present at AO languages such as AspectJ. Static weaving offers better performance than dynamic weaving. These are the trade-offs between both ASoC technologies; performance versus flexibility. It is necessary to weigh up.

AO Frameworks: Current aspect-oriented programming approaches model components and aspects as two separate entities, which are automatically weaved to produce the overall system. The resultant code is highly optimised but it is not flexible. In addition to these two entities, an aspect-oriented framework has a third entity that dynamically establishes the connections between components and aspects (see Fig. 6 and motivating papers **??**).

Having some architectural information, AO Frameworks allow the attachment of any sort of aspects to components of the specific domain. In aspect-oriented frameworks, components and aspects are developed in the same general purpose language and are composed dynamically at runtime through a middleware layer. The main goals of aspect-oriented frameworks are making explicit application architecture definition, loose coupling between components and aspects, and the possibility of COTS integration. In the approach presented by Lidia Fuentes in position paper 16, components and aspects are independent first order entities from the application domain that are weaved at run time. Aspects such as co-ordination, persistence, awareness, authentication or multiple views have been found in the collaborative virtual environment domain presented in paper 16. The framework detaches components and aspects interfaces from the final implementation classes, and they may evolve independently. Components have no knowledge about the aspects they are affected by and the number and type of aspects that are applicable to a component can change dynamically. The middleware layer is in charge of applying aspects to components though some composition information given by the software architect who specify how certain aspects must be applied to concrete components and the order in which they have to be applied.

Summarising, aspect-oriented frameworks offer the advantages of modularity, reusability, extensibility, configurability and adaptability already found in component frameworks plus separation of crosscutting concerns adopted by aspect-oriented technologies.

Aspect-Orientation and UML: Taking into consideration that crosscutting can occur in all phases of the software life cycle, we briefly discussed whether or not UML design could benefit from some ASoC technology.

Crosscutting is explicit in UML interaction diagrams. The main reason is that UML allows object-oriented system design but not aspect-oriented system modelling. Currently, there are works that extend the semantic of UML with:

– composition patterns that are used to separate the design of crosscutting requirements [24].
– new stereotypes to design well-known aspects such as synchronisation, distribution or replication, allowing thus aspect modelling at the design phase [26].

Having such designs, it is possible to automatically generate code for distinct aspect-oriented languages. With no doubt, this is an interesting and open research issue.

Acknowledgements. We would like to thank the participants of the workshop for their contributions: the position papers before the workshop, the presentations, group work and discussions during the workshop and the written reports of group work after the workshop:

Mehmet Aksit, Marie Beurton-Aimar, Adeline Capouillez, Constantinos Constantinides, Pascal Costanza, Pierre Crescenzo, Erik Ernst, Johan Fabry, Huan Fernandez, Robert Filman, Lidia Fuentes, Maurice Glandrup, Erik Hilsdale, Eduardo Kessler Piveta, Gregor Kiczales, Guenter Kniesel, Mira Mezini, Olivier Motelet, Amparo Navasa Martinez, Harold Ossher, Klaus Ostermann, Awais Rashid, Mark Skipper, Peri Tarr, Bedir Tekinerdogan, Soren Top, Eddy Truyen, Kris De Volder, Bart De Win and John Zinky,

List of Position Papers

1. Mehmet Aksit, Bedir Tekinerdogan and Lodewijk Bergmans, *The Six Concerns for Separation of Concerns*
2. Noury Bouraqadi-Saadani and Thomas Ledoux, *How to Weave*
3. Adeline Capouillez, Pierre Crescenzo and Philippe Lahire, *Separation of Concerns in OFL*
4. Yvonne Coady, Alex Brodsky, Dima Brodsky, Jody Pomkoski, Stephan Gudmundson, Joon Suan Ong and Gregor Kiczales, *Can AOP Support Extensibility in Client-Server Architectures?*
5. Constantinos Constantinides, Therapon Skotiniotis and Tzilla Elrad, *Providing Dynamic Adaptability in an Aspect-Oriented Framework*
6. Pascal Costanza, Gunter Kniesel and Michael Austermann, *Independent Extensibility for Aspect-Oriented Systems*
7. Kris De Volder, *Code Reuse, an Essential Concern in the Design of Aspect Languages?*
8. Remi Douence, Olivier Motelet and Mario Sudholt, *Sophisticated crosscuts for e-commerce*
9. Erik Ernst, *Loosely Coupled Class Families*
10. Johan Fabry, Johan Brichau and Tom Mens, *Moving Code*
11. Robert Filman, *What is Aspect-Oriented Programming, Revisited*
12. Stephan Gudmundson and Gregor Kiczales, *Addressing Practical Software Development Issues in AspectJ with a Pointcut*
13. Amparo Navasa, Miguel Perez and Juan Murillo, *Developing Component Based Systems using AOP Concepts*
14. Harold Ossher and Peri Tarr, *Some Micro-Reuse Challenges*
15. Klaus Ostermann and Mira Mezini, *Object-Oriented Composition is Tangled*
16. M. Pinto, M. Amor, L. Fuentes and J.M. Troya, *Run-time coordination of components: design patterns vs. component-aspect based platforms*
17. Eduardo Kessler Piveta and Luiz Carlos Zancanella,, *Aurelia: aspect oriented programming using a reflective approach*
18. Awais Rashid and Gerald Kotonya, *Risk Management in Component-based Development: A Separation of Concerns Perspective*
19. Mark Skipper, *Semantics of an object-oriented language with aspects and advice*
20. Soren Top, Bo Jorgensen, Claus Thybo and Peter Thusgaard, *Meta-level Architectures for Fault Tolerant Control (FTC) in Embedded Software Systems*
21. Eddy Truyen, Wouter Joosen, Bart VanHaute, Pierre Verbaeten and Bo Norregaard Jorgensen, *Customization of On-line Services with Simultaneous Client-Specific Views*
22. Bart Vanhaute, Bart De Win, Bart De Decker, *Building Frameworks in AspectJ*
23. John Zinky, Richard Shapiro, Joe Loyall, Partha Pal and Michael Atighetchi, *Separation of Concerns for Reuse of Systemic Adaptation in QuO 3.0*

References

24. S. Clarke and R. Walker. *Composition Patterns: An Approach to Designing Reusable Aspects*, In Proceedings of the 23rd International Conference on Software Engineering (Toronto, Canada; 12–19 May), pp. 5–14, 2001.
25. M. Fayad and D. Schmidt, *Object-Oriented Application Frameworks*, Communications of the ACM, 1997, 40, 10, October
26. J. L. Herrero and M. Sãnchez and F. Sãnchez, *Changing UML metamodel in order to represent separation of concerns*, Workshop on Defining Precise Semantics for UML at ECOOP'2001 (position paper), Budapest, Hungary, June, 2001

27. E. Gamma and R. Helm and R. Johnson and J. Vlissides, *Design patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995
28. David Garlan and David Notkin. Formalizing Design Spaces: Implicit Invocation Mechanisms. VDM '91: Formal Software Development Methods, pp. 31—44 (October 1991). Apears as Springer-Verlag Lecture Notes in Computer Science 551.
29. Gregor Kiczales and Erik Hilsdale and Jim Hugunin and Mik Kersten and Jeffrey Palm and William G. Griswold, 2001, *An Overview of AspectJ*, ECOOP'2001 Object-Oriented Programming, LNCS, 2072, Springer-Verlag
30. P. Tarr, M. D'Hondt, L. Bergmans & C. Lopes (eds.), *Workshop on Aspects and Dimensions of Concern: Requirements on, and Challenge Problems For, Advanced Separation Of Concerns*, to be published in the ECOOP2000 workshop proceedings, LNCS series, Springer-Verlag, 2000