Aspects, Dependencies, and Interactions

Report on the WS ADI at ECOOP'06

Ruzanna Chitchyan[1], Johan Fabry[2], and Lodewijk Bergmans[3]

[1] Lancaster University, Lancaster, UK
rouza@comp.lancs.ac.uk
[2] INRIA FUTURS - LIFL, France
Johan.Fabry@lifl.fr
[3] University of Twente, Enschede, Netherlands
lbergmans@acm.org

Abstract. For Aspect-Oriented Software Development (AOSD) the topic of Aspects, Dependencies and Interactions is of high importance across the whole range of development activities – from requirements engineering through to language design. Aspect interactions must be adequately addressed all across the software lifecycle if AOSD is to succeed as a paradigm. Thus, this topic must be tackled by the AOSD community as a whole. This first workshop, initiated by AOSD-Europe project, aimed to establish a dedicated forum for discussion of this vital topic and to attract both researchers and practitioners currently engaged with related issues. The workshop has succeeded in initiating a broad community-wide discussion of this topic and has provided an initial overview of perspectives on the state of the art as well as of outstanding issues in this area.

1 Introduction

Aspects are crosscutting concerns that exist throughout software development cycle – from requirements through to implementation. While crosscutting other concerns, aspects often exert broad influences on these concerns, e.g. by modifying their semantics, structure or behaviour. Such dependencies between aspectual and non aspectual elements may lead to either desirable or (more often) unwanted and unexpected interactions.

The goal of this first workshop was to initiate a wide discussion on dependencies and interactions between aspectual and non-aspectual elements, thus investigating the lasting nature of such dependency links across all development activities:

- starting from the early development stages (i.e., requirements, architecture, and design), looking into dependencies between requirements (e.g., positive/negative contributions between aspectual goals, etc.) and interactions caused by aspects (e.g. quality attributes) in requirements, architecture, and design;
- analysing these dependencies and interactions both through modelling and formal analysis;
- considering language design issues which help to handle such dependencies and interactions (e.g. 'dominates' mechanism of AspectJ), and, last, but not least
- studying such interactions in applications, etc.

In the following, we present the main topics discussed at the workshop and the questions risen, forming the broader view of the outstanding research issues in the aspects, dependencies, and interactions space.

2 Topics from Accepted Papers

Papers accepted to the workshop covered a broad spectrum of interaction-related problems. We have grouped these papers into related sets with each set briefly summarised below.

2.1 Requirements, Analysis and Design

This group of papers addresses the early stages of AOSD, talking about interaction identification and its impact assessment in requirements, architecture, and design.

In [6] an approach to estimating the impact of a change in requirements using the information about requirements interdependencies is outlined. The requirements dependencies are classified along 4 perspectives: temporal (e.g. Requirement R1 is before R2, etc.), conditional (e.g. R1 will apply only if R2 does), business rule, and task oriented (e.g., in order to complete R2 first need to complete R1). Furthermore, each of these can be related to another via forward, backward or parallel links; this is defined by the kind of impact triggered by a change in one of the concerns. The type of concern slices and their links are used to describe the dependencies triggered in case of a change. A weighting of such dependencies is also used to describe the severity of change. All these are then used to assess the change impact.

In [9] the focus is on concern interaction and resulting trade-off resolution issues when transiting from requirements to architecture. The functional requirements are modelled as concerns and refined into scenarios so that each scenario addresses only a single functional concern. The non-functional requirements are also represented as concerns and their refined sub concerns. Then the interactions between use cases and refined non-functional concerns are recorded and represented as a graph. From there the concerns for each scenario are classified as decisional if they are involved into trade-off making, or operational (i.e., not involved into tradeoffs). The decisional concerns for each scenario are then prioritised and these priorities are used in architecture derivation.

In [12] an approach to detecting aspect interactions in design models is presented. The system is modelled using UML class and state chart diagrams, produced separately for the core and aspectual concerns. These are then statically analysed and the potential interactions are pinpointed. After this the core and aspectual designs are woven together and some existing UML versification methods are used to verify the resultant design against behavioural properties. This approach combines light-weight syntactic analysis and formal versification of designs expressed in UML with a domain-specific state chart weaving language.

2.2 Techniques Related to the AOSD Domain

This set of papers looks at the traditionally non-AO software development methods and areas and draws a parallel between these and AOSD.

Paper [8] outlines the problems of interactions between aspects and proposes to use the work on feature dependency analysis to resolve some of them. A problem that may occur in incremental software development using AOP is first identified, namely invasive changes. The invasive change problem mainly comes from the lack of understanding of dependencies between features. To address this problem, a method is proposed that combines feature dependency analysis and AOP to provide support for incremental software development.

Paper [2] looks at how treating aspects as functions that transform programs reduces the potential aspect interactions. A simple algebraic approach that models aspects as functions is proposed, Using this approach, the paper shows that certain kinds of aspect dependencies caused by references and overlapping join points can be resolved by applying the notion of pseudo-commutativity. Two aspects are commutative if the order in which they are combined can be swapped, and pseudo-commutative if they are not commutative but can be transformed so that swapping them does not affect the program semantics The claim is made that each pair of aspects with referential dependencies or overlapping join points can be transformed into a corresponding pseudo-commutative pair.

In [13] there is an architecture outlined that separates clients, dispatchers and multiple service providers allowing for specific services to be found and used more flexibly. A dispatching intelligence between a message sender (a client) and the message receiver (a server) is discussed. This is said to provide malleability and flexible dispatch but requires the ability to specify protocols for the use of services, better semantic specification and interaction, and specification of dispatch strategies.

2.3 Language Design

This set of papers looks at the issues of AO language design and the sources of language-caused interactions.

Paper [4] presents a technique for semantic conflict detection between aspects applied at shared join points. The approach is based on abstracting the behaviour of advice to a resource-operation model, and detecting conflict patterns within the sequence of operations that various aspects apply to each resource.

In [10] the issues of how use of expressive pointcuts can cause unintended interactions are investigated. The source of the problem is that 'expressive pointcuts' specify join points based on the results of program analysis. Hence, if they are not carefully designed, the effects of weaving may depend on, or affect, the evaluation of expressive pointcuts.

Paper [5] outlines several categories of problematic interactions among structural aspects. It presents some ideas on how to avoid these situations, and it proposes the use of a logic engine for detecting certain types of aspect interactions.

3 Feature Interactions in Feature Based Program Synthesis and AOP [3]

The keynote speech: Feature Interactions in Feature Based Program Synthesis [3], delivered by Prof. Don Batory, presented an approach to represent and handle feature interaction with feature-based programme transformations. The speech invited the audience to compare and contrast aspects and features and learn how to utilise the work on feature interactions for AOSD.

Prof. Batory defined a feature as an increment in product functionality. Since these increments can often be dependent on the previously existing functionality, features will require/restrict the use of each other. He proposed that features are well suited for customisation of software products, and automated product generation can be based on tool-supported feature composition. Tools restrict the selection of features given previously selected features. For this features need to be represented as 1st class entities.

An example of such automated feature-based product composition tool was then demonstrated. In this tool the selection of features and their composition is stated in terms of grammar sentences. A domain specific language is developed for each given domain, defining the allowed and expected composition sequence of features (i.e., sentences) in that domain. Sentences that are specified by a user are checked against the domain knowledge for correct composition. Verified sentences are then used for generation of the software product in a selected implementation language (e.g., C++, Java, etc.).

Thus, automation of product generation with feature synthesis must start with domain knowledge accumulation, followed by development of feature model for that domain where each separate feature is identified and represented. Then rules are defined for required/restricted use of individual features due to their dependencies and their correct compositions. After this the product specification can be represented in terms of feature selection and composition, from which the product can be generated.

The speaker noted that in Feature-Oriented Programming (FOP), as in the functional world, programs are values and functions transform these values. If a programme is viewed as an object, the same function transformation corresponds to a method. He then drew parallels, comparing FOP and Aspect-Oriented Programming (AOP). He observed that:

- an introduction in AO is equal to an addition operation in FOP. However, the introduction addition operation of FOP is more general then AspectJ introductions as new classes, packages, etc. can also be introduced.
- FOP has an impoverished pointcut-advice language, as here heterogeneous advice and individual joinpoints form pointcuts.
- In AspectJ an advice is a function that maps a program and has an event-driven interpretation that is consistent with current AOP usage.
- AOP and FOP have different models of advising:
  - In FOP features extend individual methods by surrounding them, extending them with the new feature application. This model of advising is termed *bounded quantification*, as the advice is applied to the previously composed features only.
    - In AspectJ AOP the advising is global, since the advising is applied to the interpreter level. This model of advising is termed *unbounded quantification* as the advice applies not only the features composed before its application, but also those that will be added afterwards. The ideology of such advising implies that <u>all</u> the introductions are carried out before the advice is applied.

  Both types of quantifications are useful in different situations.

4 Discussions on Interactions and Aspects

As this was a the first workshop on the topic of aspects, dependencies and interactions, many scientific discussions focused on identifying the important questions for the domain, some of which were also discussed in more detail during the group discussions. The list of identified questions and the essence of the group discussions are summarised below.

4.1 Questions for the Research Area

Being the first forum in the area of Aspects, Dependencies, and Interactions the ADI 2006 workshop aimed to provide some initial survey of the research space, identifying and listing the topics of interest and directions for further work. Below follows the list of identified topics:

- How do we define or detect semantic interaction? How do we specify semantics of a concern?
- What taxonomy, categories, granularity and kind of interactions are there?
- What scope, binding time and binding mode do we have for composition rules? How do we do interaction detection in this context?
- How are interactions detected, propagated and resolved through different stages in the development process? What information do we need to perform this?
- Are there any interaction problems specific to the design of the pointcut language or joinpoint model.
- How can we deal with ad-hoc constraints, e.g. expert intuition?
- How does unbounded quantification combine with top-down design?
- What abstraction granularity do we need to define precedence or ordering constrains?
- How do we deal with interference, e.g. use total order or partial order?
- What language mechanisms or new operators beyond aspect precedence do we need to specify resolution?
- How do we detect interactions when an aspect suppresses a join-point needed by another aspect?
- How do we analyze aspect interactions without having the base code?
- How can refactoring techniques be used to simplify dependencies and interactions?

4.2 Group Discussions

Several of the above listed topics had repeatedly recurred at the ADI 2006 workshop. Thus, five of these recurrent topics were discussed at the group-work sessions.

4.2.1 Group 1: How Do We Define/Detect Semantic Interaction? How Do We Specify the Semantics of a Concern?

The topic of this group is an active research area; in the discussion it became clear that no profound answers to these questions could be given. Instead, several observations were made about the nature of semantic interactions and the possible forms or contexts in which semantic interactions may appear. These observations have been structured according to the following topics:

- Where do the semantic interactions occur
- What kind of properties need to be modelled for detecting interactions
- How to obtain these properties
- How do we specify the semantics of an aspect?

For semantic interactions, the focus of active research, and the group discussion, is mostly on the semantic interactions among advices. Such interactions are most common and easy to reason about at *shared join points:* join points in the program where multiple advices need to be executed. Interactions can be related to control or –shared– state, and may be influenced by the order in which advice is executed at shared join points. Detecting such interactions can be organized by iterating over all (potential) shared join points in the program. However, aspects may also interact through advices that are *not* sharing the same join point. This may be the case either because the advices have some shared state, or because one advice affects the control flow, thereby affecting the other (e.g. because some expected join point may no longer occur).

To model interactions, or design algorithms for detecting interactions, certain properties about the elements in the program must be modelled. What properties these exactly are depends on the type of interactions. Such properties need to be defined for one or more of the following elements: aspects, base code, or the woven system.

In the worst case, the software engineer must specify all properties for each program element. This yields a load for the software engineer that prohibits most practical applications. Ideally, these properties are derived automatically from the program source code. This is considered infeasible in general: because not all relevant properties can always be derived statically from a program source code, and because very quickly the size and complexity of the interaction detection grow out of hand. To this extent it is necessary to reason about a subset of all properties, for a subset of all program elements in the source code. Two approaches were discussed: firstly abstraction/focusing about the analyzed program, thereby ignoring many details, hopefully without compromising the conclusions from the detection analysis. Secondly, slicing as an alternative means to observe only a part of the complete program by leaving out many details.

Finally, the ways to represent the semantics of an aspect were discussed: here we distinguish between representing the behaviour of the aspect (i.e. of the advice), and representing the pointcuts. For representing the behaviour of advice, formal semantics such as small step semantics, state machine representations or control flow models can be used. For representing pointcuts in a precise way, possible formalisms are predicate logic or the approach followed by Event-based AOP.

4.2.2 Group 2: What Taxonomy, Categories, Granularity and Kind of Interactions Do We Have?

The first question faced by the group was whether there are any good techniques for analyzing pointcuts, but without having access to the base-level code. The goal is to be able to reason about aspects individually, in a modular fashion. However, if we would have access to the source code, then the question is if computing intersections of pointcuts would be useful enough.

The second question treated by the group was if, and how, refactoring techniques can be used to simplify dependencies among aspects, and check them separately. The analyst should not have to look at the final woven program to verify it. Instead, she/he should be able to verify the program by looking at its different aspects. It has been shown that in some cases it is possible to change the dependency order. This can be used to simplify dependencies between particularly hard to check aspects.

4.2.3 Group 3: What Scope and Binding Time/Binding Mode Do We Have for Composition Rules? How Do We Do Interaction Detection in This Context?

To get a better feeling for the concepts of scope and binding time, the group members started with defining a grid showing the possible combinations between binding time and scope. This grid was then filled in with example aspect languages or features of aspect languages with the corresponding combination. The grid is shown in Fig. 1.

	Compile Time	**Run-Time**
Static	AspectJ/Josh	CesarJ Deploy/ Association Aspects
Dynamic	Reflex/ JAC	AspectS/ Gepetto/ SteamLoom

Fig. 1. Binding Time versus Binding Mode

The discussion then focused on the need for both static and dynamic composition rules, and if these rules should be applied at the level of the aspect or each of the affected element. Participants wanted to find convincing examples of both static and dynamic cases.

For static composition rules the example found is an on-line auction site, where buyers can participate in a number of auctions. The aspects applied here are logging and access control. These need to be applied on all operations of an auction: joining the auction, placing a bid, and getting an overview of all placed bids. All attempts to join the auction need to be logged, regardless if authentication is successful or not, but to be able to place a bid, or get an overview of bids placed the buyer needs to be logged in successfully. In the former case, therefore, the logging aspect has a higher precedence than the access control aspect, and in the latter two cases, the reverse is the case.

Dynamic composition of aspects can also be encountered in this setting, albeit on the lower level of network management. All incoming network messages need to be validated for correctness, as a protection against hacking attacks. Also, incoming messages need to be logged, for traceability purposes. Two aspects are used: a validator and a logger aspect. In normal operations, the validator aspect has precedence over the logger, as only correct messages need to be logged. In case of an intensive attack by hackers, the malicious messages should however also be logged, for analysis and counter-measures. Therefore, in this case, the aspects need to be switched dynamically, whenever the validator aspect detects that an intensive attack is in progress.

4.2.4 Group 4: How Are Interactions Detected, Propagated and Resolved Through Different Stages of Development Process

The group members started by setting a basis for further discussion by agreeing that there are multiple views or models of aspects at different levels of the development lifecycle. Thus, the aspects are defined for each level of development as concerns that have a broad influence on the elements of the model at the level where the aspect is defined. They also agreed that an aspect at one level does not always need to be represented as aspect at another level. This is because a solution to a problem at one level becomes a problem to be solved at the following level. This is illustrated in Fig. 2, where a problem "Reduce operational costs for car dealers" stated at requirements will be represented as, for instance, a requirements template (solution), this template will then need to be represented as a part of architecture design (problem), leading to creation of multiple alternatives architectures and finally a choice of one of them (solution). The chosen architecture solution (problem for Design) will have to be elaborated into a detailed design solution, etc.

With the above view, we can define the requirements -> architecture -> etc. as a chain of problems and solutions. In this case, if a model is created for each level, we can use the model transformation techniques to move from one level to another. With the correct transformations we would be able to preserve the semantics of aspects. Moreover, if decisions on interaction resolution for the higher level are made, and a 1-to-1 transformation is appropriate, these interaction resolutions can be used at the lower level as well. Otherwise in case of 1-n mapping, we need to optimize the transformations from multiple perspectives.

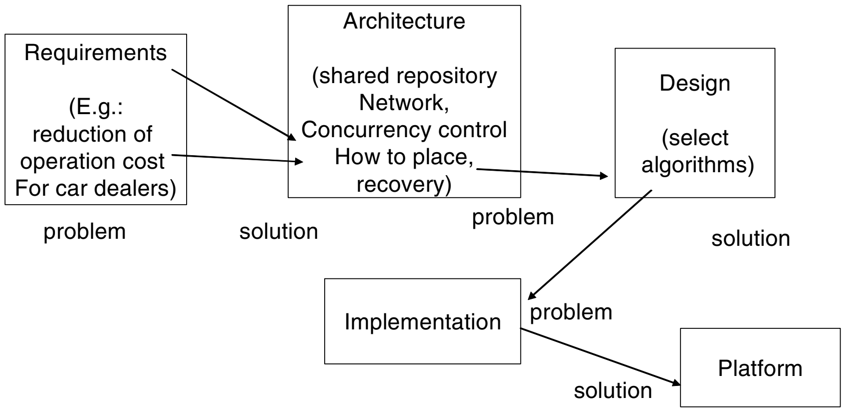
Requirements
(E.g.: reduction of operation cost For car dealers)
problem
Architecture
(shared repository Network, Concurrency control How to place, recovery)
solution
problem
Design
(select algorithms)
solution
Implementation
problem
Platform
solution

Fig. 2. The Problem-Solution Chain for Aspect Modelling

The information about interactions should be obtained from domain knowledge, past experience (similar to design patterns), e.g. selection of a decisions (e.g. topology is related to response time). But more research needs to be done to clarify and define as to what details do we need to know in order to realize these transformations.

The group participants carried out an initial brainstorming to answer the above questions, and suggested that some information that could be used is:

- The call graph with call probability can be used to estimate the performance of some communication at different levels (e.g., collect the statistics of car dealers, simulate the algorithms with the given statistics, put the priorities from the statistical information into the architectural trade-offs)
- If we know the values for the relative importance of concerns in requirements, these can be passed to components and connectors to preserve the same constraints in the architecture.
- Prepare and reuse trade-off catalogues for various concerns and their mapping

4.2.5 Group 5: Are There Specific Problems to Pointcut Languages in Terms of Aspect Interactions?

The group participants thought that most problems of interactions in pointcut languages are particularly related to system evolution. Such problems are, for instance, that:

- non-local edits to the code results in changes of the set of matching join points. This in turn can result in aspect interactions, making two aspects inadvertently picked up due to code modification being applied to the same join points.
- pointcuts can be based on runtime types. This for instance, may result in *proceed* changing the runtime types, and *arguments* can check runtime types. An example of this is presented in Fig. 3.
- aspects can modify the type hierarchy (e.g., declare parents), etc.

```
after(Float f) returning: args(f) && ... {}
around(Number n) : execution(foo) && args(n)
    { proceed(new Double(0.5)); }
void foo(Number n) { ... }
```

Fig. 3. An Example of Pointcut-Based Aspect Interaction

- As a result of this discussion, the group participants concluded, that the pointcuts, indeed are most likely to cause aspect interactions. Thus, they reasoned that new kinds of joinpoints should be developed that leverage this interaction-causing characteristic rather than try to neutralize it. Such joinpoints may intentionally promote desired aspect interactions. For instance, advice execution at such a point may enable an aspect which in turn may disable other aspects, or alike.

The group participants also discussed weather static joinpoints have lesser interaction-related problems than the dynamic ones. They agreed that both static and dynamic joinpoints are equally likely to cause interactions, though in the case of static joinpoints such interactions could be resolved in the implementation of the compilers.

5 Do Aspectual Dependencies and Interactions Prevent Modular Reasoning?

A panel of experts from the AOSD-Europe project discussed the question *Do aspectual dependencies and interactions prevent modular reasoning?* Each of the four panellists presented a specific view on the issue, which are briefly outlined below.

5.1 Panel Positions

Adrian Colyer questioned how could an approach that gives better modularity not improve modular reasoning? This is a contradiction. He pointed out that any new modularity mechanism, by virtue of allowing new ways of modularity, will have to provide new ways of composition. So if the aspectual modules and compositions "break modular reasoning", then what kind of reasoning is broken? He argued that it is the "old" modularity that will be broken, but a new form of modular reasoning will arise from new modular representation and composition. Adrian drew a parallel of the AOSD acceptance with that of past paradigm shifts by recounting that before procedures the developers had needed to do global reasoning. With procedures they could do local reasoning, but still had global data. With objects they could also locally define data. Aspects are the next step in this ladder of evolution, allowing to locally define functionality and data used in multiple locations by objects or procedures.

Adrian summarised the argument of AO opponents as "if we need to know about all aspects to reason about a module, how can we have modular reasoning"? He then explained that, in his view, such argument means that modular reasoning implies complete locality. However, modular reasoning is not "all or nothing", but a mechanism to support partial reasoning about partially localised code or modules in the same way as, for instance, procedures allow useful analysis about the procedure content, but they do need to refer to the instance variables. Thus, AOP supports better partial modular reasoning.

Awais Rashid stated that aspects are about abstraction (abstracting away from the details of how that aspect might be scattered and tangled with other functionality), modularity (allowing to modularise the details of interest, hence facilitating modular reasoning), and composability [11]. He noted that in his view composability is of a paramount importance as modules need to relate to each other in a systematic and coherent fashion. Aspects provide advancement in composition mechanisms, thus complementing the traditional modular reasoning with compositional reasoning, i.e., the ability to reason about global and emergent properties of the system using the modularised information. Awais underlined that in his view, the important contribution of AOP is the support for this compositional reasoning, which helps developers to more effectively reason about dependencies and interactions compared to traditional software engineering techniques. Hence, aspects do not prevent modular reasoning as this is supported via the abstraction and modularity support. Furthermore, the composability properties of aspect-oriented techniques enhance the compositional reasoning support available to developers.

Theo D'Hondt suggested that maybe such questions as the topic of this panel were rising because the AO community has approached the notion of aspects from the wrong perspective. Theo pointed out that historically Computer Science has developed by increasing the level of abstraction of the main elements used in development. However, this progression trend was broken in case of aspects. The break was in that historically each new theory had worked to show that the newly proposed abstraction (e.g., procedure, object) is the "rule" for the development in that suggested paradigm. However, in case of AO, aspects are treated as "an exception" to the existing abstractions. Thus, Theo asked if the AO community should try and view objects and hierarchical decomposition as a "specialisation" of aspects? He proposed that, aspects should be more general abstractions than those of the previous theory. In conclusion, Theo suggested that aspects should be studies "on their own" as modules, along with their reasoning support. However, this will only be possible if aspects are viewed as more general, subsuming previous abstraction types, so the view on aspects should be revisited.

Mario Sudholt was sceptical that modularity and full obliviousness were possible with aspects. Mario explained that traditionally, when writing a system, one would start with modules and interfaces and when writing them he or she would have a good knowledge of the program structure and behaviour. However, with AOP, as it is today, the development is moving from this modular development, into the style of evolutionary development. He concluded that he does not believe that the current state of art provides any proof that AO supports an adequate modularity. Current approaches to modular properties in the presence of aspects (e.g.,[1, 7]) are either too restrictive or not powerful/systematic enough to render a solution. He suggested that this point will remain to be seen in the future.

5.2 Discussion

The discussion turned to the question of what is the simplest mental image of aspect. It was suggested that in most cases aspect is understood as "interrupt routine plus interrupt processing". However, this is a very primitive view and the community should work to move away from this primitive image of aspect to more appropriate image for a fully fledged module. Indeed, an aspect has state and behaviour, as a class has state and methods. A class has called methods running over variables, and aspect has pointcuts, advice and state, with advice running when the pointcut is matched. So aspect is as much a module as a class.

The workshop participants also agreed that one should not think that "an aspect belongs to a class". Aspects are for modular reasoning about the aspects themselves, not the class. The claim that aspects change the classes in an unintended way can be equally well stated about the classes changing each other as well. The claim that the class does not know which aspects will apply on it, is also valid about the call to procedures: one does not know which procedure will be called, particularly in cases of late binding.

Adrian Colyer also argued that successive levels of modular reasoning are possible with AspectJ, looking at the aspect only; then at aspects with the type of applying advice, i.e., before, after or around; after this the details about the body of the advice can be included into the reasoning. Thus, different levels of detail for reasoning can be available at the different element consideration views.

The discussants talked about global versus local design with aspects versus OO. It was argued that in OO the global design is hard but the local effects are very clear. On the other hand, with AO global design is clear and relatively easy, though its effects on local reasoning are more difficult to account for. Tool support can help in assisting local reasoning by completing the local details recovered from global compositions. However, no tool support can adequately recover the global details from local ones, supporting global reasoning. Thus, AO seems to progress along the right path.

Additionally, the participants agreed that aspects bring into programming languages the notions of quantification over temporal and conditional references which are broadly used in natural language (e.g., notions about history of the programme execution, etc.). Thus aspects enrich the current programming languages.

After a lively discussion, in summary, most workshop participants agreed that aspect interactions need to be addressed, but they do not, in principle, prevent modular reasoning.

6 Conclusion

This AOSD-Europe workshop on Aspects, Dependencies and Interactions provided an opportunity for presentations and lively discussion between researchers working on AOSD and dependencies and interactions in general from all over the world. As a first workshop on the topic, it has established a list of important questions which will stimulate and guide the further research in this area. We intend to continue the work stared at this event in the future years by organising a series of follow up workshops.

7 Workshop Organisers and Participants

7.1 List of Organizers

- Ruzanna Chitchyan: Lancaster University, UK (Co-chair).
  Email: rouza@comp.lancs.ac.uk
- Johan Fabry: Vrije Universiteit Brussel, Belgium (Co-chair).
  Email: johan.fabry@vub.ac.be
- Lodewijk Bergmans: Universiteit Twente, The Netherlands.
  Email: L.M.J.Bergmans@ewi.utwente.nl
- Andronikos Nedos: Trinity College Dublin, Ireland. Email: nedosa@gmail.com
- Arend Rensink: Universiteit Twente, The Netherlands.
  Email: rensink@cs.utwente.nl

7.2 List of Attendees

The list of attendees officially registered for the workshop is presented below (a number of unregistered attendees also participated, but are not listed):

1. Mehmet Akşit (University of Twente, The Netherlands)
2. Tomoyuki Aotani (University of Tokyo, Japan)
3. Don Batory (University of Taxes at Austin, USA)

4. Lodewijk Bergmans (University of Twente, The Netherlands)
5. Walter Cazzola (DICo Università di Milano, Italy)
6. Ruzanna Chitchyan (Lancaster University, UK)
7. Adrian Colyer (Interface21, UK)
8. Theo D'Hondt (Vrije Universiteit Brussel, Belgium)
9. Remi Douence (INRIA, France)
10. Pascal Dürr (University of Twente, The Netherlands)
11. Johan Fabry (Vrije Universiteit Brussel, Belgium)
12. Safoora Khan (Lancaster University, UK)
13. Kwanwoo Lee (Hansung University, Korea)
14. Jose Mango (Instituto Politecnico de Leiria, Portugal)
15. Hidehiko Masuhara (University of Tokyo, Japan)
16. Sonia Pini (DISI, Università di Genova, Italy)
17. Awais Rashid (Lancaster University, UK)
18. Pouria Shaker (Memorial University of Newfoundland, Canada)
19. Pablo Sánchez (University of Malaga, Spain)
20. Frans Sanen (Katholieke Universiteit Leuven, Belgium)
21. Daniel Speicher (University of Bonn, Germany)
22. Maximilian Störzer (Universitat Passau, Germany)
23. Mario Südholt (INRIA, France)
24. Éric Tanter (University of Chile, Chile)
25. Tim Walsh (Trinity College Dublin, Ireland)

References

[1] J. Aldrich and C. Chambers, "Ownership Domains: Separating Aliasing Policy from Mechanism", ECOOP'04, 2004, LNCS, 3086, pp. 1-25.

[2] S. Apel and J. Liu, "On the Notion of Functional Aspects in Aspect-Oriented Refactoring", in Aspects, Dependencies, and Interactions Workshop (held at ECOOP), Lancaster University Computing Department Technical Report Series, ISSN 1477447X. Lancaster, 2006, pp. 1-9.

[3] D. Batory, "Feature Interactions in Feature-Based Program Synthesis", University of Texas at Austin, Dept. Computer Sciences, Austin, Technical Report TR-06-52, September 2006.

[4] P. Durr, L. Bergmans, and M. Aksit, "Reasoning about Semantic Conflicts between Aspects", in Aspects, Dependencies, and Interactions Workshop (held at ECOOP), Lancaster University Computing Department Technical Report Series, ISSN 1477447X. Lancaster, 2006, pp. 10-18.

[5] B. Kessler and É. Tanter, "Analyzing Interactions of Structural Aspects", in Aspects, Dependencies, and Interactions Workshop (held at ECOOP), Lancaster University Computing Department Technical Report Series, ISSN 1477447X. Lancaster, 2006, pp. 70-76.

[6] S. Khan and A. Rashid, "Analysing Requirements Dependencies and Change Impact Using Concern Slicing", in Aspects, Dependencies, and Interactions Workshop (held at ECOOP), Lancaster University Computing Department Technical Report Series, ISSN 1477447X. Lancaster, 2006, pp. 33- 42.

[7] G. Kiczales and M. Mezini, "Aspect-oriented programming and modular reasoning", 27th International Conference on Software Engineering (ICSE 2005), 2005, ACM, pp. 49-58.
[8] K. Lee, "Using Feature Dependency Analysis and AOP for Incremental Software Development", in Aspects, Dependencies, and Interactions Workshop (held at ECOOP), Lancaster University Computing Department Technical Report Series, ISSN 1477447X. Lancaster, 2006, pp. 62-69.
[9] J. Magno and A. Moreira, "Concern Interactions and Tradeoffs: Preparing Requirements to Architecture", in Aspects, Dependencies, and Interactions Workshop (held at ECOOP), Lancaster University Computing Department Technical Report Series, ISSN 1477447X. Lancaster, 2006, pp. 43-52.
[10] H. Masuhara and T. Aotani, "Issues on Observing Aspect Effects from Expressive Pointcuts", in Aspects, Dependencies, and Interactions Workshop (held at ECOOP), Lancaster University Computing Department Technical Report Series, ISSN 1477447X. Lancaster, 2006, pp. 53-61.
[11] A. Rashid and A. Moreira, "Domain Models are NOT Aspect Free", MoDELS/UML, 2006, Springer LNCS, 4199, pp. 155-169.
[12] P. Shaker and D. K. Peters, "Design-level Detection of Interactions in Aspect-Oriented Systems", in Aspects, Dependencies, and Interactions Workshop (held at ECOOP), Lancaster University Computing Department Technical Report Series, ISSN 1477447X. Lancaster, 2006, pp. 23-32.
[13] T. Walsh, D. Lievens, and W. Harrison, "Dispatch and Interaction in a Service-Oriented Programming Language", in Aspects, Dependencies, and Interactions Workshop (held at ECOOP), Lancaster University Computing Department Technical Report Series, ISSN 1477447X. Lancaster, 2006, pp. 19-22.