On Composition and Reuse of Aspects

Jörg Kienzle, Yang Yu, Jie Xiong
School of Computer Science
McGill University
Montreal, QC H3A 2A7
Canada
contact: Joerg.Kienzle@mcgill.ca

Abstract

This position paper investigates the possibilities of separation, modularization and reuse offered by aspect-orientation, concentrating not on the technical or syntactic problems, but on the inherent issues resulting from inter-aspect dependencies. An aspect is defined based on the services it provides, on the services it requires and on the services it removes from other aspects. A classification of aspects is established based on the way they interact with each other and on the way their functionality is triggered. Composition rules and the weavability criteria are defined based on this classification. Moreover, the impact of the dependencies of aspects on the level of achievable reuse is analyzed. Finally, the paper shows how the general ideas apply to the aspect-oriented programming environment AspectJ.

1 Introduction

Separation of concerns is a fundamental principle of software engineering that in its most general form refers to the ability to identify, encapsulate, and manipulate those parts of software that are relevant to a particular concept, goal, task, or purpose. The benefits of a successful modularization of concerns during the implementation phase are obvious: simpler code structure resulting in improved readability of program code, program code that is easier to customize and adapt to new situations, increased possibilities for reuse.

In order to help developers, software development methods, e.g. the Unified Process [1], define a step-by-step process that leads the development team from an initial requirements document through to the final implementation [2]. Most approaches start by analyzing the system requirements based on use cases, which capture the expectations that the final users of the software may have. In a sense they focus on the different concerns of the end-users. During the design phase however, most approaches concentrate on elaborating an object-oriented design, i.e. decomposing the system into objects, each of them providing a well-defined part of the main functionality of the system. As a result, secondary functionality, e.g. distribution support, is often poorly encapsulated. This phenomenon is known as the “tyranny of the dominant decomposition” [3], and aspect-orientation [4] might be a possible way to counter it.

Of course, the notion of main functionality is relative. It has never been precisely defined, but is usually used to denote what is particular to a certain piece of software. These days, general mechanisms, for instance mechanisms that deal with concurrency and failures, distribution, or security, are considered secondary functionality.

This classification into main and secondary functionality, often also referred to as *functional* and *non-functional aspects*, is very unfortunate. It somehow conveys the feeling that there are important concerns and less important concerns during the development of a piece of software, which often leads to the mistake that only the functional part of an application is developed following software engineering principles, and the non-functional part, e.g. fault tolerance, is added later on.

It is our firm belief that there are no such things like non-functional aspects. Every concern of a certain piece of software is important, is part of its functionality. During design all concerns must be considered and integrated in order to obtain an elegant solution.

What are called non-functional aspects are actually concerns that are more general, i.e. they are likely to be present in other applications as well. Of course, it is tempting to separate these aspects from the other functionalities of the application and make then “generic”, meaning that modularize them in such a way that they can be easily reused in other contexts and applications. The idea is very legitimate and it does not take long to convince any sensible programmer that such a separation would be great. Aspect-orientation might just be the right way to achieve this kind of separation.

This paper investigates the possibilities of separation and reuse offered by aspect-orientation, concentrating not on the technical or syntactic problems, but on the inherent issues resulting from inter-aspect dependencies. Section 2 defines the essence of an aspect based on the services it

provides and on the services it requires from other aspects. Section 3 classifies aspects according to the way they interact with each other. Based on this classification, Section 4 provides composition rules for aspects and Section 5 examines reusability issues. Section 6 illustrates how the presented ideas apply to one of the main-stream aspect-oriented development environments, AspectJ. Section 7 takes a closer look at circular dependencies. Section 8 presents recommendations for aspect developers, and the last section summarizes the results of this work.

2 Aspects

For the subsequent discussion, it is important to specify clearly what we mean when we talk about an aspect.

From our understanding, an aspect at the design and implementation level is a main abstraction that encapsulates that part of the design solution that addresses a certain concern expressed at the analysis level. On on hand, the aspect provides a certain functionality: it implements the concern. We'll designate the set of the services it provides *P*. Services can be seen as the entry points or interface offered to the rest of the system. On the other hand, the aspect may depend on functionality offered by other aspects. The set of services it depends on is named *D*. Optionally, an aspect might remove functionality of other aspects. The set of services it removes is named *R*. Obviously, *R* is a subset of *D*. An aspect is therefore categorized by the three sets *P*, *D*, and *R*.

What is needed to accurately describe a *service* is intentionally left open. On one end, specifying the complete semantics of an aspect is a challenging task, and out of the scope of this paper. One the other end, object-oriented programming languages often just use method signatures to specify their interface to the outside world. Applying this idea to aspects would mean specifying the signatures of *P*, *D*, and *R*.

If we want to use UML to depict an aspect, we might be tempted to use the representation of a class or interface. Unfortunately, these constructs only show what a component provides to the environment, and not what it requires from others. UML stereotypes make it possible to extend the base UML concepts and add additional meaning to them. Fig. 1 shows an `<<aspect>>` stereotype with three new compartments: *Provides*, *Depends On* and *Removes*. It is not clear if an aspect should be seen as an extension of the UML class, of the UML package or rather an extension of the UML collaboration. Discussions are still in progress [5, 6].

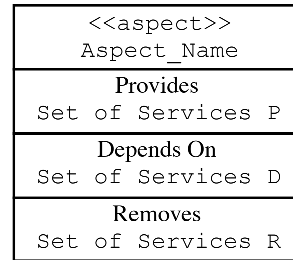

Fig. 1: UML Representation of an Aspect

3 Aspect Interaction

In this section we attempt to classify aspects according to the way they interact with each other. We have established two classification criteria: the *activation mechanism* and the *dependencies*.

3.1 Activation Mechanism

The activation mechanism of an aspect is determined by analyzing *when* the aspect delivers its functionality. There are two different kinds: *autonomous* and *triggered*.

Autonomous: An aspect is autonomous if it can act on its own, i.e. it does not need to be stimulated to deliver its functionality. It typically performs its duties continuously or periodically.

Triggered: Initially, a triggered aspect is passive. It waits for some other part of the application to activate it, and only then it delivers its service.

This classification is similar to the one found in object-orientation, where one distinguishes active and passive objects [7]. Active objects act autonomously, whereas passive object must be triggered, i.e. only execute methods when they are called from the outside.

Of course this classification is not absolute. An aspect may provide autonomous services and triggered ones, similar to the Time-Triggered Message-Triggered Objects presented in [8], which provide periodically executing services as well as services triggered by messages. For the sake of clarity, however, such mixing of activation mechanisms is discouraged.

3.2 Dependencies

We essentially distinguish three different kinds of dependencies: *orthogonal*, *uni-directional* and *circular*.

The functionality that an *orthogonal* aspect provides to an application is completely independent from the other functionalities of the application. The only thing it might depend on is activation time (see Section 3.1 above), or general application-independent information provided by the run-time environment, e.g. information on the virtual machine, current method name, etc.

Unfortunately, such aspects are not very common. An example of an autonomous orthogonal aspect is a clock counter. Every second, the counter is increased by one. No explicit triggering is needed (probably the counter is implemented using an independent thread or interrupts), and there are no shared data structures between the clock aspect and others. Measuring the time elapsed between two events can be seen as a triggered orthogonal aspect. One might think that there is a semantic dependency of such a timing aspect on the part of the application it actually performs timing on. This is, however, not true. The dependency is only on the fact that the aspect has to be triggered twice: once to start the timing, and a second time to stop it.

One of the most popular orthogonal triggered aspects is logging. For debugging purposes, a logging aspect can be applied to various places in an application to print out stack traces, etc.

A *uni-directional* aspect depends on some functionality (service or data) offered by other aspects in the application. Without this functionality, it can not deliver its services. Among uni-directional aspects, we can further distinguish between *uni-directional preserving* and *uni-directional modifying* ones.

Uni-directional preserving aspects provide new services based on services of other aspects, but do not alter or hide the other services in any way. The properties and functionalities of the other aspects are preserved.

[9] presents an example of two triggered uni-directional preserving aspects. It describes an aspect-oriented implementation of a telecom application that handles phone calls. In order to set up correct billing, the elapsed time of long distance phone calls must be measured. The long distance timing aspect uni-directionally depends on the call aspect, adding timing information to the calls. It is not orthogonal, because it has to associate timing with calls, and therefore depends on the existence of the call aspect. The billing aspect in turn depends on the call and the timing aspect, for it has to know the calls source and destination city, and the elapsed time in order to calculate the total cost. In the same context one can imagine an aspect that periodically collects statistical information on long distance calls, e.g. the average length of calls. This is an example of an autonomous uni-directional preserving aspect that depends on the call and the timing aspect.

A *uni-directional modifying* aspect replaces or modifies functionality of some other part of an application, but it does this transparently; the other aspect is not aware of this, and therefore does not have to behave differently. In a sense, a uni-directional modifying aspect wraps around or encapsulates some services provided by other aspects. As a result, some of the original services might not be provided anymore.

As an example of a uni-directional modifying aspect, imagine a typical banking application. Some banks (at least Swiss banks) allow good clients to overdraw their account. Clients with a bad credit history on the other hand are not be allowed to do this. The desired effect can be achieved by encapsulating in one aspect the account behavior, and design an additional aspect that denies withdraw requests in case of insufficient funds. The additional aspect removes the withdraw service from the account aspect.

Circular dependency is the strongest form of dependency. It occurs when several aspects are mutually dependent. The simplest form is encountered when two aspects depend on each other, i.e. the first aspect requires some service provided by a second aspect, which, in turn, can only deliver its service with the help of the first one. Another way of looking at this from the perspective of an aspect that you are adding to an application is the following: if in order to make the overall application work with the new aspect it is necessary to modify other aspects, then there is circular dependency.

An example of a circular aspect has been presented in [10]. In this example, a transaction aspect is added to a previously non-transactional application, allowing the application to deal with concurrency and failures. The aspect itself provides the run-time support for transactions, making it possible to execute methods transactionally. However, the application must state which method calls it wants to make transactional, and what actions should be taken in case a transaction aborts due to a failure.

Circular-dependent aspects are so tightly coupled that one might argue that it makes no sense to consider each aspect separately. This first impression will be confirmed when considering composition and reuse later on. It is often simpler to treat them as a single aspect. The set of services the single aspect provides is the union of the services the individual aspects provide, and likewise for the set of services it depends on and the set of services it removes. In the following sections we do not consider circular-dependent aspects, they will be revisited in section 7.

The following table summarizes the classification established in this section:

Class of Aspect	Restriction
Orthogonal	$D = \emptyset$
Uni-directional preserving	$R = \emptyset$
Uni-directional modifying	no restriction

Table 1: Classification of Aspects

4 Composition Rules

In AOP, the so-called *aspect weaver* composes the different aspects to form the final application. This composi-

tion can be done statically, i.e. at compile-time, or even dynamically during the execution of the application. Implementing such an aspect weaver is far from trivial, and there are lots of technical issues that need to be addressed when composing aspects. In this section, however, we will concentrate on the more fundamental problems of aspect composition. Even though a set of aspects might be technically composable, it might be conceptually impossible.

In order to simplify the discussion, we introduce the notion of an *aspect group*. Aspects in an aspect group all have some dependency relationship. An executable application consists of at least one non-empty aspect group, containing at least one autonomous aspect. Initially, the set of aspect groups that forms the final application is empty. Step by step, additional aspects are added. The set of aspect groups that form the final application is called a *configuration*.

If we represent aspects as nodes, and dependencies as directed edges, the representation of a configuration takes the form of a directed acyclic graph (short DAG) as shown in Fig. 2. We'll call it the *configuration dependency graph*. Each *component* of the dependency graph forms an aspect group.

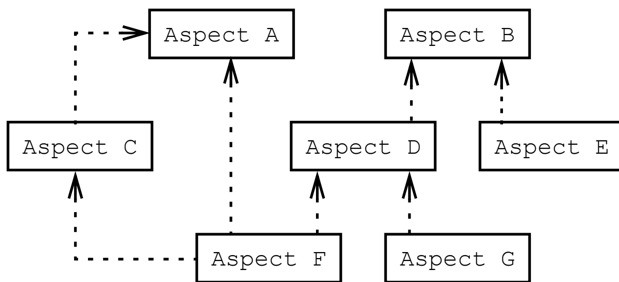

Fig. 2: A Configuration Dependency Graph

The composition rules for aspects in this section is based on the classification presented in the previous section.

Orthogonal aspects are very flexible — due to their orthogonality there are no restrictions on composing such aspects with others. When adding an orthogonal aspect to the final application, a new aspect group is created, i.e. a new component is added to the graph.

Uni-directional aspects must be added to an already existing aspect group. The set of services that the aspect requires must be provided by the aspects that are already in the group. It is also possible to combine aspect groups, i.e. join previously separated components of the graph, in order to obtain the required set of services.

4.1 Weavability

An interesting problem is the *weavability problem*, i.e. determining if a given set of aspects can be composed in such a way that all service requirements are fulfilled.

In graph theory, this is equivalent to solving a multi-commodity flow feasibility problem [11] with additional node constraints. The graph to be analyzed contains one node for each aspect and is fully connected. Every type of service will be considered a separate flow. An aspect that provides a certain service is a source for the flow (provides one flow unit). An aspect that removes the service is a sink (consumes one flow unit). Aspects that depend on the service are mandatory transshipment nodes for the corresponding flow. They can be modeled by an additional constraint that states that the sum of incoming flows for this node must be equal to one.

If and only if there exists a feasible flow, then the application is weavable. By calculating the flow distribution that uses the lowest number of arcs, and then inversing all arcs, we obtain the dependency graph of the final application.

5 Making Aspects Reusable

One of the major encouragements for using AOP is reuse. After having identified a certain concern, the idea is that AOP should allow one to modularize and implement this concern in an aspect. Later on, this aspect should be usable in every application that exhibits the need for the concern. Again, there are technical issues that must be solved in order to make aspects reusable, e.g. how to specify the required, provided and removed services in a concise way. In this section, however, we will concentrate on the obstacles introduced by aspect dependencies.

An even stronger form of reusability is *genericity*. What we want to achieve in this case is to write an aspect in such a way that it can be added to an application without disturbing the already existing structure. In other words we want to add support for a certain concern to an application just by adding the aspect that implements the concern to the configuration.

The difficulty of providing such a form of reusability increases depending on the class of aspect.

Orthogonal aspects can be reused in any context. They are generic per se. They do not depend on any other aspects, and therefore do not remove any existing services. They do not disturb any existing aspect group configuration, since they always start a new group. In the dependency graph, orthogonal aspects will show up as sinks. In Fig. 2, `Aspect A` and `Aspect B` are orthogonal aspects.

Uni-directional preserving aspects also make good candidates. Since they do not remove any services, they can be added to any aspect group that provides the required ser-

vices. Of course, when moving a uni-directional preserving aspect from one configuration into a new one, any aspects it depends on must be either moved as well, or equivalent services must already be available in the new configuration. In the dependency graph, new uni-directional aspects shows up as a source nodes. For instance, in Fig. 2, the uni-directional aspects `Aspect E`, `Aspect F` or `Aspect G` might just have been added to the configuration.

Uni-directional modifying aspects are hard to reuse, since they modify the services of aspects they depend on. They can only be added to an aspect group if it remains weavable, i.e. the new aspect does not remove services that are needed by other aspects.

6 AOP Mechanisms

This section analyses the support of the concepts presented above provided by AspectJ [12], one of the mainstream aspect-oriented programming environments.

6.1 Interface Specification

Somehow, aspect-oriented programming environments must provide a means for specifying what services an aspect provides, what services it depends on, and what services it removes. This has been an area of research for a long time, and elegant solutions to this problem still have to be found.

AspectJ takes the Java approach. The services provided by a class or aspect are determined based on Java visibility rules. Inside visible code, all potential joinpoints are advisable, meaning that they can be used as triggers or points of extension for adding additional behavior.

There is no special part where dependencies are specified. An aspect potentially depends on all other modules that are visible or that it imports. By looking closely at the code, the services it actually uses can be determined.

The services that an aspect modifies or removes are very hard to determine. Potential candidates are the destinations of `around` advice, but also `before` and `after` advice that modify the behavior of the class or aspect they are advising.

6.2 Activation Mechanism

Just as conventional object-oriented environments, aspect-oriented development environments support autonomous aspects. The autonomy of aspects is typically implemented by the underlying operating system. Autonomous aspects are either separate processes, or implemented based on threads.

AOP is however particularly well suited for implementing triggered aspects. They are usually activated by the aspect-oriented run-time, which in turn is stimulated by intercepting some specific event.

In AspectJ, for example, pointcut designators allow a developer to specify when an aspect is to be activated. For instance, it is possible to intercept calls to / and execution of methods, throwing and handling of exceptions, and reading and writing of fields. It is also possible to activate aspects based on control flow information.

6.3 Aspect Semantics

There are several mechanisms that allow an AspectJ programmer to write uni-directional aspects.

First of all, aspects are subject to the same visibility rules as normal Java classes. They can call methods, or read from / write to fields depending on their respective mode (public, protected, private) and the package they belong to. As soon as an aspect makes an explicit reference to some other class or aspect, a dependency is created.

Next, aspects can use static *introduction* to add new fields or methods to classes or aspects at compile-time. If explicit names are used, then again a dependency is created. However, the introduction mechanism allows a programmer to use pattern matching rules to defer the destination of the introduction to weave-time.

Finally, aspects can add code *before* or *after* any joinpoint defined in the code they are advising. It is even possible to wrap code *around* a joinpoint, optionally replacing the code that would have been executed at this point.

6.4 Composition

At some point, AOP environments must perform the weaving, i.e. composing all aspects of an application to yield the final application. Logically, the composition ordering is determined by the configuration dependency graph. In order to obtain a possible sequential composition order, topographical sort [13], also known as linear extension, can be applied to the configuration dependency graph.

In current aspect-oriented environments, the dependency graph information is in general encoded by the developer in a separate configuration language, or in the aspect language itself.

The latter is true for AspectJ. The pointcut designators in an aspect specify the set of joinpoints to which the advice must be applied. If several advice apply to the same joinpoint, then the developer can specify an ordering among them by using the *dominates* primitive. If some aspect A is specified to dominate some aspect B, then advice in A take precedence over advice in B. In a sense, A wraps around B (or B is nested in A). In this case, the execution order of the advice is:

- before advice in A

- before advice in B
- original code at joinpoint
- after advice in B
- after advice in A

If around advice are used, the ordering takes the following form:

- around advice in A
  (optional around advice in B
  (optional original code at joinpoint))

6.5 Drawing the Line for Dependencies

As we have seen in the previous section, in order to achieve high reusability or even genericity, a developer of an aspect should strive for low dependencies. As strange as it might seem, dependency does not only depend on the nature of the problem, but also on the power of the weaving mechanism. Surprisingly, some dependencies can be replaced by exploiting the activation mechanism in a clever way.

To illustrate this idea, consider a typical bank account, implemented as an aspect. In addition, there is a security policy that states that the account balance should not drop below zero. This policy is implemented in a separate security aspect. At first one might think that the security aspect is uni-directionally dependent on the account aspect, for it must monitor all changes to the state of the balance of the account.

It turns out that this is not necessarily true. The security aspect can be turned into a orthogonal one that prevents any numerical value to drop below zero. It is the weaving mechanism that links it to the account, i.e. activates the security aspect on every change of state of the account balance.

In AspectJ, for instance, the account would be implemented as a normal Java class with a balance field. The security aspect would be implemented as an aspect containing a before advice that verifies that the field value is higher than the amount passed to the withdraw method. The dependencies on class fields can be removed by using AspectJ run-time information. The triggering of the aspect, i.e. intercepting every write access to the balance attribute of the account, is done in the pointcut definition.

7 Circular Aspects Revisited

After having examined composition, reuse and support mechanisms we can now reexamine circular-dependent aspects. Several reasons push to believe that they should be considered a single aspect:

- Composition: When composing an application, a set of circular-dependent aspects must be added to an configuration as a whole. Moreover, it is only possible if the configuration provides the union of the services needed by each circular-dependent aspect.
- A set of circular-dependent aspects can only be reused as a group. The added services are the union of all services provided by the circular-dependent aspects.
- During the weaving process, a set of circular-dependent aspects must conceptually be woven at the same time. This may actually lead to implementation problems, similar to the problems encountered when compiling mutually dependent source files.

However, in certain situations it might make sense to consider them separately, e.g. if one aspect of the group is fairly generic. This is the case in the previously mentioned example [10], where transaction support has been implemented as a separate aspect in AspectJ. Transactions are a generic concept that can be applied to parts of an already existing application. However, the application wants to be aware of this, especially when a transaction aborts due to some underlying failure. In this case, the application might want to try the same transaction again, or decide to perform some alternative computation, and / or inform the user of the failure, etc. As a result, the application and the transaction aspect are tightly coupled.

What we suggest in this case is to try and extract that part of the application aspect that deals with transactions and make it a separate aspect. As a result, the two circular dependent aspects (application and transaction) can be transformed into an orthogonal and two uni-directional ones (the application without transaction handling, transactions, and the application-specific transaction handling part). This is illustrated in Fig. 3. The feasibility of such a transformation again depends heavily on the expressiveness and power of the weaving support. If this can be achieved for the transaction example using AspectJ remains to be explored.

Of course, programmers must be aware that the new application aspect and the application-specific transaction handling aspect have very tight semantic dependencies, although physically separated. Modifying the application aspect most probably also requires modifying the transaction handling one.

8 Discussion

As we have seen in the previous sections, the dependencies of an aspect have a profound impact on the ways it can be composed with others and on the possibilities of reuse.

Therefore, when developing an aspect that is to be made reusable or even generic, a programmer should first determine the semantic nature of the concern that is to be modu-

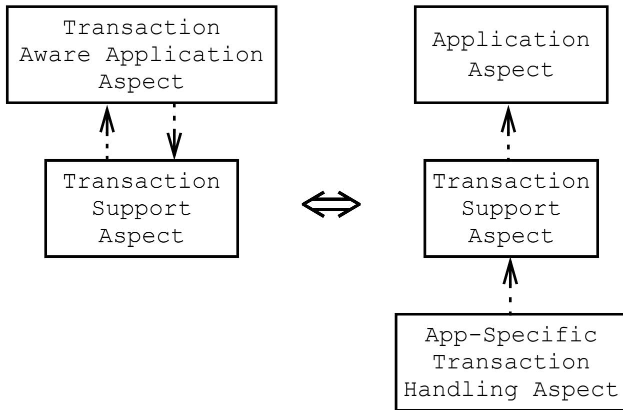

Fig. 3: Transforming Circular-dependent Aspects into Uni-directional Ones

larized. This will provide a hint on what degree of decoupling might be achievable. Next, the developer should try and classify the aspect according to the rules mentioned above.

Orthogonal aspects are most flexible, followed by uni-directional preserving ones. Uni-directional modifying aspects are not easily reusable, since adding them to an application may compromise weavability. Circular-dependent aspects should, if possible, be transformed into several uni-directional ones.

The weaving mechanism offered by the aspect-oriented environment has also an important impact on the dependencies. If it is not powerful enough, or the weaving language is not expressive enough, then additional dependencies might be artificially introduced into the system. On the other hand, exploiting the power of the aspect weaver and aspect run-time information might make it possible to remove dependencies. Imagine an aspect that monitors some data, and triggers some action if the data changes. Such an aspect would fall into the autonomous uni-directional category, since it is dependent on the data it monitors. However, if the aspect run-time allows activation of aspects based on data changes[1], then the dependency can be removed from the aspect. In a sense, the dependency is re-introduced later at weave-time, when the actual configuration is assembled. As a result, the monitoring aspect now is triggered and orthogonal, and hence can be reused in a straightforward way.

Based on these observations, we encourage designers of aspect-oriented programming environments to conduct further research in this direction. For instance, the decision for adding new features such as new pointcut designators to AspectJ should be based on whether or not such a new feature would make it possible to remove a certain kind of dependency.

1.AspectJ, for instance, allows triggering aspects when a field of a class is modified.

9 Conclusion

In this position paper we have investigated the possibilities of separation, modularization and reuse offered by aspect-orientation in general.

We have defined an aspect based on the services it provides, on the services it requires from other aspects and on the services it removes. Furthermore, a classification of aspects has been established. Aspects can be *autonomous* or *triggered*, depending on the activation mechanism. The dependencies lead to a categorization into *orthogonal*, *uni-directional preserving*, *uni-directional modifying*, and *circular-dependent* aspects. The influence of the power of the weaving mechanism on dependency has been highlighted.

Composition rules have been established based on these criteria, and the notion of weavability has been defined based on flow feasibility analysis. Likewise, the impact of the semantic nature of aspects on the level of achievable reuse has been analyzed.

Finally, we have presented how the general ideas of this paper apply to the aspect-oriented programming environment AspectJ, and made recommendations for determining the usefulness of new features.

10 Acknowledgments

The authors would like to thank the anonymous reviewers of the FOAL and SPLAT workshop committees for their detailed comments.

11 References

[1] Jacobson, I.; Booch, G.; Rumbaugh, J.: *The Unified Software Development Process*, Addison Wesley, Reading, MA, USA, 1999.

[2] Hutt, Andrew T. F.: *Object Analysis and Design – Description of Methods*. Object Management Group, John Wiley & Sons, Inc., 1994.

[3] Tarr, P. L., et al.: “N Degrees of Separation: Multi-Dimensional Separation of Concerns”. In *Proceedings of the 21st International Conference on Software Engineering (ICSE’1999)*, pp. 107-119, IEEE Computer Society Press / ACM Press, 1999.

[4] Elrad, T.; Aksit, M.; Kiczales, G.; Lieberherr, K.; Ossher, H.: “Discussing Aspects of AOP”. Communications of the ACM 44 (10), pp. 33 – 38, October 2001.

[5] First International Workshop on Aspect-Oriented Modeling with UML. Held at the *First International*

Conference on Aspect-Oriented Software Development, April 22-26, 2002, Enschede, The Netherlands.

[6] Second International Workshop on Aspect-Oriented Modeling with UML. Held at the *Fifth International Conference on the Unified Modeling Language - the Language and its Applications*, September 30 - October 4, 2002, Dresden, Germany.

[7] Briot, J.-P.; Guerraoui, R.; Lohr, K.-P.: "Concurrency and Distribution in Object-Oriented Programming", *ACM Computing Surveys **30**(3)*, September 1998, pp. 291 – 329.

[8] Kim, K.H.; Masaki, Ishida; Liu, Juqiang: "An Efficient Middleware Architecture Supporting Time-Triggered Message-Triggered Objects and an NT-based Implementation". In *Proceedings of the second IEEE CS International Symposium on Object-Oriented Real-time Distributed Computing (ISORC'99)*, pp. 54 - 63, St. Malo, France, May 1999.

[9] The AspectJ Team: *The AspectJ Programming Guide*, Xerox Corporation, February 2002.

[10] Kienzle, J.; Guerraoui, R.: "AOP — Does it make sense? The case of concurrency and failures". In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, pp. 37 - 54, Malaga, Spain, June 2002, Lecture Notes in Computer Science **2374**, Springer Verlag, 2002.

[11] Cook, W. J.; Cunningham, W. H.; Pulleyblank, W. R.; Schrijver, A: *Combinatorial Optimization*. John Wiley and Sons, Inc. 1998.

[12] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersen, M.; Palm, J.; Griswold, W. G.: "An Overview of AspectJ". In *Proceedings of the 15th European Conference on Object–Oriented Programming (ECOOP 2001)*, pp. 327 – 357, June 18–22, 2001, Budapest, Hungary, 2001, Lecture Notes in Computer Science **2072**, Springer Verlag, 2001.

[13] Aho, A. V.; Hopcroft, J. E.; Ullman, J. D.: *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, USA, 1987.