

# On the Criteria of Aspectual Component Models

Steven Op de beeck, Johan Grégoire,  
Eddy Truyen and Wouter Joosen  
Department of Computer Science  
Katholieke Universiteit Leuven

Celestijnenlaan 200A, B-3001 Leuven, Belgium

{steven.opdebeeck,johan.gregoire, eddy.truyen,  
wouter.joosen}@cs.kuleuven.be

## ABSTRACT

In this position paper, we explore the thesis that Aspectual Component Models promote software engineering properties such as analyzability, predictability, maintainability, evolvability, management of semantic interactions etc. We propose a set of criteria for evaluating new and existing aspectual component models. We illustrate the proposal by applying the criteria to AspectJ, CaesarJ, JAsCo and CAM/DAOP. We believe that defining such a set of criteria is an important step towards the inception of an optimal aspectual component model and the necessary language support.

## 1. INTRODUCTION

This paper builds upon the concept of *Aspectual Components* as defined in [6], however explaining the details behind the concept is beyond the scope of this paper.

Aspectual Components (ACs) were introduced as a construct for reconciling object-based and aspect-based decomposition. ACs represent concerns that are expressed in terms of their own modular structure. The modular structures of different ACs do not necessarily fit, but rather crosscut each other. ACs are specified by an interface that consists of an expected and provided facet. The expected facet of the interface represents the abstract join points referred to from the aspectual component. These abstract pointcuts are resolved separately, inside a so called *connector* construct. A connector essentially maps the modular structure of one AC to the other. This separation of connecting logic from the actual component logic allows more reuse, easier extensibility and better modularity.

In order to be able to evaluate an Aspectual Component Model (ACM) we propose an initial set of evaluation criteria. These criteria emphasize important concepts that we believe, should be supported by an ACM. Each concept influences certain Software Quality Factors (SQFs) that we would like an ACM to achieve. SQFs are sometimes referred to as “ilities”: reusability, modularity, evolvability, reliability, etc. The proposition of criteria is based on some empirical re-

search that was conducted and is open for discussion.

This paper is structured as follows. In Section 2 an overview is given of the criteria for Aspectual Component Models. Section 3 then illustrates the defined criteria by taking a look at AspectJ, CaesarJ, JAsCo and CAM/DAOP. In Section 4 we briefly discuss related and future work and we end this paper in Section 5 with a conclusion.

## 2. CRITERIA

In this section we discuss the criteria for Aspectual Component Models and we explain their influence on SQFs. The criteria that we will discuss here are concerned with these topics: Aspectual Collaborations, Interaction, Integration and Deployment.

### 2.1 Aspectual Collaborations

*Support for the decoupling of implementation and binding parts and A clear interface that specifies this separation using expected and provided facets.*

The idea is based on the concept of Aspect Collaboration Interface (ACI), that was introduced in CaesarJ [7]. It improves upon similar concepts from the work of *Aspectual Components*. An ACI plays a central role by splitting an aspectual component up in an expected and a provided facet. The *provided* facet defines which behaviour the aspectual component will provide itself in order to realise its purpose in the application context where it is deployed. This facet will be implemented in the aspect implementation. The *expected* facet on the other hand contains the details about the integration of the AC into the application context, it defines what the aspectual component expects from the application in order to realise what it must provide itself. This facet will be resolved by the aspect binding, which is essentially the connector construct from [6].

We argue that an aspectual component model should support a separation into provided and expected facets as in the ACI model. In component terms this boils down to an interface that relates both parts to the whole, but also a mechanism to realise the parts independently and allow them to be reconnected as specified by the ACI.

The collaboration interface provides a detailed description of the aspectual component, making it easier to plug aspectual components in and out of different base applications, hereby increasing reusability of the component implementation. Splitting up an aspectual component in disjoint parts—one that resolves the expected facet and another that supplies the provided facet—increases modularity, allowing for

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AOSD.06 SPLAT, March 20–21, 2006, Bonn, Germany.

separate development and increasing support for the component based approach.

### *Advice declaration and Implementation.*

*The declaration of advice in the AC interface and Support for the separation of advice and pointcuts.*

An important concept in aspect oriented programming (AOP) is the advice type-declaration (before, after, around, ...) and implementation. All AOP approaches have the concept of advice in one form or another. In the context of an ACI the advice and its type-declaration can be placed at different locations. In their work on aspect-aware interfaces Kiczales, et al. [4] (section 4.2, par. 3) argue that the type of advice should be included in the aspect's interface. In a similar way we argue that the AC interface is an appropriate location to declare the advice-type because it helps in describing the component's purpose and changing an advice's type is much more significant than changing its implementation. Additionally, the advice implementation should be a part of the AC implementation and not of the binding, as this is an essential piece of the behaviour of an aspectual component. This means that advice is completely separated from the pointcuts that bind it, making it easier reusable like the rest of the aspect implementation.

Of course, separating advice from its pointcuts means that some way is required to reconnect them later on. Mind that this is completely different from the problem of connecting the AC to other (Aspectual) Components. In many AOP approaches this is actually tackled by the concept of pointcuts itself, which will be discussed in section 2.3. The issue of reconnecting advice and pointcut is something that is specific to aspectual collaboration and its separation of implementation and binding.

The issue can be handled in a number of ways: In middleware-oriented approaches the reconnection of advice and pointcuts is tackled by an external deployment descriptor that actually defines the pointcuts and links them to the appropriate advice. However, since this requires external reference to the advice by the deployment descriptors, it is usually<sup>1</sup> necessary for advice to be declared in the interface of the AC. Approaches that are more language-oriented introduce concepts like *aspect*, *pointcut*, *advice(type)* and *connector (binding)* into the language. The AC implementation provides some pointcut abstraction that is bound to the advice. A connector is used to refine these pointcut abstractions into concrete pointcuts. A possible disadvantage of this technique is that the connector module requires recompilation after each modification.

However, this is no strict separation, it is possible that a middleware-oriented approach introduces a number of language constructs to assist the developer, and vice-versa.

Another difference between both approaches is the way in which they provide access to the joinpoint context from within the advice. Middleware approaches often use parameter-passing<sup>2</sup> and reflection to inspect the joinpoint context. Whereas language approaches provide constructs that abstract away from the details of information passing. However, the choice for one mechanism or the other isn't actually that crucial, what matters is that information passing is handled in a uniform or generic way. The manner in which advice retrieves information about (or supplies information to) the context

it is applied to, should be independent of any particular context. It is up to the binding to realise the context specific connection (see section 2.3).

### *Behavioural Contracts.*

*Support for the extension of AC interfaces with behavioural contracts.*

Behavioural contracts can be regarded as a useful extension of the current AC interface. Contracts are important in a CBSD setting. While it is true that an ACI model already allows the composition of some sort of contract, there is a need for more detailed contracts. The contract specified through an AC interface is limited to the definition of the expected, provided and possibly advice methods, but not their behaviour. When deploying a third-party aspectual component within an application, it is important to know how certain advice influences the normal flow and state of the application in order to guarantee the correctness of existing behaviour.

That's where behavioural contracts come in. They aim at capturing the behaviour of methods in a well-defined and readable fashion. And unlike the source code of components, contracts are usually available to the users of the component.

Supporting DbC hence increases SQFs such as correctness, modularity and reusability.

### *Generic Interface and Implementation.*

*Support for generics in the AC interface and implementation.*

By using generics it becomes possible to define aspectual components in a more abstract, a more generic way. Instead of working with very specific types, generics can be used to abstract away those specifics. It is only when integrating the aspectual component in a concrete environment that these generic types will be filled in with concrete types.

Generics are an enabling concept for achieving a high degree of flexibility and reusability of aspectual components.

## 2.2 Interaction

*Support for a mechanism that allows some control over specific aspectual interactions.*

It is not always an easy task to determine which aspects interact. Aspects do not necessarily need to be applied to the same join point to witness the effects of some interaction. Generally speaking, when a certain aspect alters the behaviour (or structure) of a component it may be interacting with another aspect that was depending on the behaviour (or structure) as it was in its unaltered state. Aspect interaction can be regarded as a form of *feature interaction* [10] limited to the interaction of aspect behaviour. These interactions can also be classified as *intentional* or *unintentional*. Each of these categories leads to *positive* or *negative* side-effects —depending on the effect being desirable or undesirable. It is important to keep in mind that the problem of feature interaction is far from being solved, so far even that no mechanism can be expected that has complete control over aspect interactions. Consequently, it is not our intention to state that some approaches have complete control of the interaction problem, while some have not. Rather, we will classify approaches according to the expressiveness of the mechanism and its ability to solve some subset of interaction problems.

Some approaches enable the declaration of precedence between the execution of aspects' behaviour at a specific joinpoint. While more advanced discrete techniques allow the

<sup>1</sup>depending on the invasiveness of the model.

<sup>2</sup>`public Object trace(JoinPointContext jpc)`

execution of an aspect to depend on a condition, like, whether or not a certain other aspect is executed, or a certain property holds.

In very small systems, with only a few simple aspects, it might be possible to determine if the system is interaction-free (specifically, free of negative interactions). However, with rising complexity of the system, detecting and identifying aspect interactions becomes even more complex. Even when existing mechanisms in Aspectual Component approaches restrict control to discrete points in the execution—the joinpoints—it remains a complex task. In practice the type of control is mostly limited to handling intentional interaction between aspects (aspects designed to cooperate), and correcting a specific undesirable interaction, that was determined by tracing back some undesirable side-effect to that interaction.

SQFs like reliability, security, extensibility, evolvability are affected by the ability to control aspect interaction. Reliability depends on the confidence in the software that it does what it is expected to do. Because unintentional interactions or mostly unexpected and—with rising complexity—even unpredictable, reliability suffers. Extensibility and evolvability are both concerned with adding new functionality, possibly new aspects, which may lead to new interactions.

## 2.3 Integration

*Specification mechanism for integration and Support for uniform access to contextual behaviour.*

Integration deals with the possible ways of connecting the aspectual component with other (aspectual) components—sometimes referred to as the base. In the ACI model the binding is responsible for connecting the AC to this base. The binding contains a specification of the composition (like pointcuts) that describes exactly how integration occurs. As mentioned earlier (section 2.1), this can take the form of an external configuration description that contains the pointcuts, represented as rules, policies or descriptors. Another form is that of a first-class language construct, a module that defines pointcuts which also represented by some language element.

Most ACs define advice behaviour that requires specific information about the context it is applied to. Instead of hard-coding the paths to access this data into the advice—making it hard for the advice to be used in another context—it makes more sense to provide a mechanism for uniform access to the required information. The expected facet of the ACI realises just that. It provides an interface for retrieval of context data by the AC implementation. In the binding, the behaviour for retrieval of the information for that specific binding context is realised. The same is true when advice needs to supply information to the context.

Two popular techniques of realising the expected facet of the interface in the binding are intertype declarations and the wrapper principle. The wrapper principle is generally preferred over the other. It allows the modular definition of functionality outside the component being wrapped—the wrappee. A wrapper can introduce new behaviour and state to adapt the wrappee to the expected interface.

Affected SQFs are reusability and modularity. Decoupling the binding part from the AC implementation enhances the AC's modularity and enables it to be reused by making it easier to bind to any other base. By enabling easy and uniform access to context information, it allows enhanced reuse of AC implementation.

## 2.4 Deployment

*Support for dynamic deployment of ACs.*

Before aspects are active in a certain environment they need to be *deployed*. Once deployed the aspects' advices will be applied at the specified times—before, after or instead of certain method invocations. Deployment can take place in a static or dynamic way. All approaches offer at least static deployment. Static deployment means that the aspect is active from the moment the system starts running and remains active until the system stops running. Dynamic deployment on the other hand allows activation and deactivation of aspects while the system is in a running state.

The addition and removal of ACs is something that is not completely covered by the concept of deployment, but nevertheless an important and useful feature of an ACM.

Dynamic deployment increases changeability as it allows for a system to be reconfigured at runtime by activating and deactivating aspectual behaviour, also enhancing a system's adaptability to a runtime context.

## 3. EVALUATION

In this section we illustrate the criteria by evaluating three approaches: AspectJ<sup>3</sup> [3], CaesarJ [1], JAsCo [11] and CAM/DAOP [8, 9, 2].

For each of the criteria we determine the approaches' support for it, if they support it at all. Table 1 gives an overview of the evaluation.

### 3.1 Aspectual Collaboration

*Decoupling Implementation and Binding.* In AspectJ this can be realised by using aspect inheritance. The aspect implementation declares abstract pointcuts and implements the provided facet and the advice. A subclass declares concrete pointcuts and realises the expected. Inheritance has the disadvantage that the binding class references the implementation class, resulting in an incomplete decoupling and a dependency that impedes binding reusability. JAsCo suffers from same problem because the connector contains a direct reference to the implementation, however refinement classes allow for the expected facet to be separated from the connector (binding), greatly alleviating this problem.

CaesarJ allows the complete decoupling of the implementation and the binding part, and the external reconnection by the weavlet.

CAM/DAOP allows a complete decoupling of implementation and binding. The deployment descriptor (binding) connects aspect advice, which is made available through an *evaluated* interface (declaration of messages the aspect is able to evaluate), to service calls and events between components.

*A clear interface that specifies this separation using expected and provided.* Abstraction in AspectJ and refinement in JAsCo can be used to declare the interface and separate the declaration of the provided and expected methods. CaesarJ has explicit support for this separation of facets. However, none of the approaches enforce this separation.

CAM/DAOP has clear interfaces describing what behaviour is expected from other components or aspects. However there

<sup>3</sup>Extensions exist that add behavioural contracts to AspectJ. For this evaluation we assume AspectJ is extended by Pipa [12].

Criterion	AspectJ	CaesarJ	JAsCo	CAM/DAOP
<b>Aspectual Collaborations</b>				
- decoupling implementation and binding	+	++	++	++
- AC interface with expected and provided	+	+	+	+
- advice type-declaration in interface	-	+	+	-
- separation of advice and pointcuts	+	-	++	++
- <i>behavioural contracts</i>	+	.	.	.
- <i>generic interfaces and implementation</i>	++	.	.	.
<b>Interaction</b>				
- controlling aspectual interaction	+	+	++	+
<b>Integration</b>				
- specification mechanism for integration	pointcuts	pointcuts	pointcuts	XML descriptor
- uniform access to context behaviour	+	++	++	++
<b>Deployment</b>				
- dynamic deployment	-	+	++	++

**Legend:**

++	well supported	-	crit	
+	supports basics		<b>bold:</b>	topic
-	lack of support		<i>italics:</i>	valued extension
.	not available			

**Table 1: Evaluation overview**

is no interface that declares what should be internally provided in the aspect.

*Advice type-declaration in the interface.* We argued that it is possible to introduce implementation and binding decoupling in AspectJ and JAsCo using class/hook-inheritance. However, AspectJ does not allow advice specialization or abstract advice and hence does not support advice declaration in the interface.

In CaesarJ and JAsCo, advice type-declarations can be placed in the interface.

CAM/DAOP has a clear interface that declares what it is able to advise, the type of this advice, however, is not included in this declaration but decided upon during the binding.

*Separation of advice and pointcuts.* In CaesarJ, advices are placed within the binding, together with the pointcuts. But as argued in the definition of this criterion, we believe that implementing advice within the binding is not optimal. AspectJ and JAsCo allow advice to be placed in the implementation and use pointcut abstraction to realise the connection to pointcuts later on.

CAM/DAOP clearly separates advice implementation from the pointcuts in the aspect binding. The aspects have no information, at any time, about how they will be composed. This information is described in the deployment descriptor and only available to the platform itself.

*Behavioural Contracts.* Only when AspectJ is extended by Pipa [12] —a formal Behavioural Interface Specification Language (BISL) for AspectJ— does it have support for behavioural contracts.

CaesarJ, JAsCo and CAM/DAOP have no support for behavioural contracts.

*Generics.* *Generics* are at this point only supported by AspectJ —starting from version 1.5.0 (AspectJ 5). It provides support for generic and parameterized types within point-

cuts, inter-type declarations and even declare statements. Parameterized types may freely be used within aspect members. There is even support provided for generic abstract aspects.

CaesarJ, JAsCo and CAM/DAOP offer no support for generics.

### 3.2 Interaction

*Aspect Interaction.* AspectJ and CaesarJ offer only basic support for interaction by allowing the arbitrary ordering of multiple aspects without providing an appropriate location to place this declaration —the precedence declaration becomes even crosscutting.

JAsCo goes one step further by allowing aspect ordering and combination-strategies, defined within the connector that deploys interacting aspects. This allows aspects to be ordered, disabled or enabled in certain situations.

CAM/DAOP allows control over the order in which aspects are applied. This is described in the deployment descriptor.

### 3.3 Integration

*Specification mechanism for integration.* AspectJ, CaesarJ and JAsCo all use language-based pointcuts to connect to the base.

CAM/DAOP uses a deployment descriptor in XML-format to connect to the base. These are no ordinary pointcuts, as in the language-oriented approaches, but they are semantically comparable.

*Uniform access to context behaviour.* In order to adapt a base-object to the expected interface, AspectJ offers the concept of introductions (inter-type declarations) that allows the extension of existing classes with extra behaviour and state. But as argued in the criterion definition, the wrapper principle is preferred over the introduction approach.

CaesarJ and JAsCo both offer the mechanism of wrapping

base-objects —albeit each in a somewhat different way. CAM/DAOP offers properties as a concept to realise context dependencies between aspects and components. A property provides a homogeneous mechanism for sharing information between components (including aspects).

### 3.4 Deployment

*Dynamic Deployment.* AspectJ only allows static deployment, while both CaesarJ and JAsCo allow static and dynamic deployment. Additionally, JAsCo supports the addition and removal of ACS at runtime.

CAM/DAOP employs a runtime-weaving mechanism, that realises the connections between aspects and components during execution. It is also possible to adapt the behaviour by adding, removing or modifying the connection information.

## 4. RELATED AND FUTURE WORK

This paper can be seen a first attempt at defining a set of criteria for evaluating ACMS. Future work may refine this set of criteria to better grasp the concepts that are important to these ACMS.

To the best of our knowledge there has not been any previous work attempting this definition of criteria. Many new models have been proposed, but there seems to be no clear view as to what properties a model should possess and why.

The criteria presented in this paper have all been covered to some degree in one ACM or another. However, it would be interesting to have a single model that supports all of them. It can be argued that, for example, the behavioural contracts criterion does not necessarily need to be an integral part of an ACM. It could be seen as an extension that can be applied to an existing model. For example, a possible way to support behavioural contracts in JAsCo might be to define something similar to Pipa which can later then be applied to JAsCo. This is similar to the Java Modeling Language [5], an extension to Java.

## 5. CONCLUSION

In this paper we have proposed a set of criteria that represent important properties of ACMS. We discussed the influence of these criteria on SQFS. Once the set of criteria was defined we used them to perform an example evaluation of four existing platforms (AspectJ, CaesarJ, JAsCo, CAM/DAOP) in which we explained to what degree the platforms supported each criterion. We also briefly discussed related and future work.

## 6. REFERENCES

- [1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. Overview of caesarj. In *Transactions on Aspect-Oriented Software Development*, 2006.
- [2] L. Fuentes, M. Pinto, and P. Sanchez. Dynamic weaving in cam/daop: An application architecture driven approach. In *Proceedings of the Dynamic Aspect Workshop (DAW05) at AOSD05*, 2005.
- [3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353. Springer-Verlag, 2001.
- [4] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ACM SIGSOFT International Conference on Software Engineering (ICSE 05)*, Mar. 2005.
- [5] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, 2000.
- [6] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, Mar. 1999.
- [7] M. Mezini and K. Ostermann. Conquering aspects with caesar. In M. Aksit, editor, *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages pp. 90–100. ACM Press, 2003.
- [8] M. Pinto, L. Fuentes, and J. M. Troya. Daop-adl: an architecture description language for dynamic component and aspect-based development. In *GPCE '03: Proceedings of the second international conference on Generative programming and component engineering*, pages 118–137. Springer-Verlag New York, Inc., 2003.
- [9] M. Pinto, L. Fuentes, and J. M. Troya. A dynamic component and aspect-oriented platform. *Comput. J.*, 48(4):401–420, 2005.
- [10] E. Pulvermuller, A. Speck, J. Coplien, M. D'Hondt, and W. D. Meuter. Feature interaction in composed systems. In *ECOOP '01: Proceedings of the Workshops on Object-Oriented Technology*, pages 86–97, London, UK, 2002. Springer-Verlag.
- [11] D. Suvée, W. Vanderperren, and V. Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29. ACM Press, 2003.
- [12] J. Zhao and M. C. Rinard. Pipa: A behavioral interface specification language for aspectj. In *FASE*, pages 150–165, 2003.