

A Study of Aspect-Oriented Design Approaches

Steven Op de beeck Eddy Truyen Nelis Boucké
Frans Sanen Maarten Bynens Wouter Joosen

Report CW435, February 2006



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

A Study of Aspect-Oriented Design Approaches

*Steven Op de beeck Eddy Truyen Nelis Boucké
Frans Sanen Maarten Bynens Wouter Joosen*

Report CW435, February 2006

Department of Computer Science, K.U.Leuven

Abstract

This report carries out a study that is based on literature of the state-of-the-art in Aspect-Oriented Design Approaches. The goal of this work is to position these design approaches within the full life-cycle of a software engineering process that encompasses requirements analysis, architecture design, detailed design, implementation, maintenance, evolution, etc. The study takes on the form of an evaluation that is based on a set of criteria communicating the goals of this report. The report provides a description and classification of these criteria including a discussion on the influence they have on various software quality factors. A conclusion will present a number of interesting observations about the current state-of-the-art.

Keywords : Aspect-Oriented Software Development, Aspect-Oriented Design Approaches, Separation of Concerns, Software Quality Factors.

CR Subject Classification : D.2.2, D.2.13, C.4

AMS(MOS) Classification : Primary : 68N99.

A Study of Aspect-Oriented Design Approaches

Steven Op de beeck, Eddy Truyen, Nelis Boucké¹,
Frans Sanen, Maarten Bynens, Wouter Joosen

Distrinet
Department of Computer Science
K.U. Leuven
Celestijnenlaan 200A
B-3001 Heverlee, Belgium

{steven.opdebeeck, eddy.truyen, nelis.boucke, frans.sanen,
maarten.bynens, wouter.joosen}@cs.kuleuven.be

February 3, 2006

¹Ph.D. scholarship funded by the “Institute for the Promotion of Innovation through Science and Technology in Flanders” (IWT Vlaanderen).

Abstract

This report carries out a study that is based on literature of the state-of-the-art in Aspect-Oriented Design Approaches. The goal of this work is to position these design approaches within the full life-cycle of a software engineering process that encompasses requirements analysis, architecture design, detailed design, implementation, maintenance, evolution, etc. The study takes on the form of an evaluation that is based on a set of criteria communicating the goals of this report. The report provides a description and classification of these criteria including a discussion on the influence they have on various software quality factors. A conclusion will present a number of interesting observations about the current state-of-the-art.

Contents

Introduction	1
<i>The Goal of this report</i>	2
<i>The Methodology</i>	2
<i>Aspect-Oriented Design</i>	2
1 Comparison Criteria	4
1.1 Definitions	5
1.1.1 Compositional Separation	5
Classification of Approaches	6
<i>Element and Relation symmetry.</i>	6
<i>Element and Relation asymmetry.</i>	8
<i>Element symmetry and Relation asymmetry.</i>	8
<i>Element asymmetry and Relation symmetry.</i>	10
1.1.2 Compositional Expressiveness	12
Classification of Approaches	13
<i>Binding, Integration & Reconciliation.</i>	13
<i>Join Points and Pointcuts.</i>	13
<i>Aspectual Collaboration Connectors.</i>	14
1.1.3 Alignment to Phases	14
Classification of Approaches	15
<i>Aligned to Requirements.</i>	15
<i>Aligned to Implementation.</i>	15
<i>Aligned to Requirements and Implementation.</i>	16
<i>Aligned to None.</i>	16
1.1.4 Refinement Mapping	16
Classification of Approaches	16
<i>Extending</i>	16
<i>Creating</i>	17
<i>Both</i>	17
1.1.5 Traceability of Design	17
Classification of Approaches	19

CONTENTS

	<i>External Traceability “$\mathbf{R} \rightarrow \mathbf{D} \rightarrow \mathbf{I}$”</i>	19
	<i>Internal Traceability</i>	20
1.1.6	Process Support	20
	Classification of Approaches	21
	<i>Design Process</i>	21
	<i>Design Guidelines</i>	21
2	Approach Evaluation	23
2.1	Classification Overview	23
2.2	Discussion	25
2.2.1	Compositional Separation	25
2.2.2	Compositional Expressiveness	28
2.2.3	Alignment to Phases	31
2.2.4	Refinement Mapping	33
2.2.5	Traceability of Design	34
2.2.6	Process Support	36
	Conclusion	38
A	Overviews	41
A.1	Criteria	41
A.2	Approaches	44
A.3	AOD and UML	45
	Bibliography	46

Introduction

Aspect-Oriented (AO) deals with the *modularisation* and *encapsulation* of *crosscutting concerns*. A concern represents something of importance to a stakeholder. It can encompass multiple requirements during analysis, but a concern must also be designed and implemented. A concern has different representations across multiple phases. The term crosscutting refers to those concerns that cannot be effectively modularised using well-known OO development techniques. As such, they cut across the structure of concerns that **can** be captured by these existing paradigms.

Typical concerns that crosscut originate from all kinds of non-functional requirements, e.g. synchronization, persistence, optimization, adaptation (also called software quality factors (SQFs)). But the phenomenon is not limited to non-functional requirements, even functional ones can have their behavioural logic spread out over multiple modules. AO tries to resolve this matter by enabling the modularisation of a crosscutting concern's data and behaviour.

Separating these concerns into modules means they need to be reconnected again using some composition mechanism. Composition essentially controls where and when concern behaviour is applied, how concerns get along in order to avoid unexpected interaction, even how “to be composed” structures are made conform.

The focus of Aspect-Oriented has initially been on the implementation phase of the software development life-cycle. This is the reason AO is generally associated with Aspect-Oriented Programming (AOP)[Kiczales et al., 1997] —being an extension¹ or something placed on top of Object-Oriented Programming.

An approach to *software engineering* that supports AO is Aspect-Oriented Software Development (AOSD). AOSD provides techniques for systematic identification, representation, modularisation and composition of these crosscutting concerns. Initial work in AOSD was about developing languages, frameworks and tool-support for the implementation phase (AOP). In order to become *systematic*, the development process must also address these concerns at the *analysis* and *design* phases. This gives rise to the term *Early Aspects* [Rashid et al., 2006] at the analysis phase and *Aspect Oriented Modelling* during design. So, rather than viewing AOP as a synonym to AOSD, it should be considered an integral part.

¹AO forms a second-order extension of any programming paradigm. As such, AO does not compete with existing techniques, it should be seen as an extension.

The Goal of this report is to perform a study of the state-of-the-art in aspect-oriented design (AOD) approaches and to position these approaches within the **full life-cycle of a software engineering process**. We want to investigate the potential of these approaches in the field of **product-line engineering of large-scale systems**. A recurring concept is that of **process support**, an important requirement for any approach will be the support for a well defined design process.

Our intention is to develop an understanding on the numerous AOD approaches that exist, to learn what differentiates them and, finally, to select those approaches or techniques that best satisfy the views and goals that have been put forward.

The Methodology we followed boils down to a comprehensive literature study of AOSD-Europe's [Chitchyan et al., 2005] "Survey of Aspect-Oriented Analysis and Design Approaches" and the individual research papers referenced therein. This is an initial study, based solely on literature. This work does not attempt to perform an evaluation based on an in-depth case-study of the approaches.

During our study, a set of criteria was drafted, related to the criteria in the survey, adjusted to match the goals in this report. Based on the study and discussion of the approaches, a classification was defined for most of the criteria. This classification was used to categorise the approaches.

Each of the classes of these criteria has an influence on the software quality factors of the designed system. Every class-definition will contain a discussion on the impact of the class to some important SQFs, that were determined to be the most relevant ones for the criterion that class belongs to.

Aspect-Oriented Design is based on well-known design methodologies. Since the notion of AO originated at the programming level, it seems justified to question its relevance at the earlier phases, particularly at design time. Regular design (ie. UML) already permits the use of multiple diagrams or views that contribute to the same software construct (structure, sequence, communication, state transition, etc.). Can now be safely concluded that SoC has since long been solved in design? When taking a closer look, it becomes clear that this is not the case. Communication diagrams do cut across the class structure of the system, but what they don't do is produce untangled views on the concerns within that system. This should become clear when asking yourself why the design cannot be cleanly mapped onto an untangled implementation. Indeed, because the design wasn't untangled to begin with.

We feel that design should be regarded as a process that starts from an abstract model, inferred from the result of a previous phase, that is recursively refined into a more detailed design, until it can be easily mapped onto an (aspect-oriented) implementation technology.

All this can be seen as an indication that some abstraction is needed that represents a (crosscutting) concern at the design level, that can be easily mapped onto aspects in an AOP language. Since the design phase lies in between of the requirement and implementation phases, it would be desirable that this

concern abstraction also reversely maps onto the concerns identified during the requirement phase. This mapping between phases is called *traceability*, which is an important characteristic that has a beneficial influence on qualities like maintainability, evolvability, etc. (see section 1.1.5).

Adding a new unit of decomposition to a design language requires the definition of both syntax and semantics for this unit and for its composition specification that integrates it into the model as a whole —counterparts to implementation-level bindings like, for example, pointcuts in AspectJ [Kiczales et al., 2001].

At every level within the development process, it should be possible to distinguish abstractions that represent concerns. This greatly facilitates the communication by means of a model or a series of models, as people can view the system at the level of detail they require, whether they are trying to understand the high-level design or are implementing a specific module, they're still all looking at the same system with the same compositions, though at different levels of abstraction.

This report is structured as follows. Chapter 1 explains the comparison criteria used throughout this work. Chapter 2 will conduct an evaluation based on these criteria and try to compare the approaches accordingly. Chapter 2.2.6 will present a conclusion of the analysis.

This report is an **AspectLab**² deliverable for the *Aspect-Oriented Software Engineering Lab* that groups the research efforts that consider aspects throughout all stages of the software development process.

²<http://ssel.vub.ac.be/aspectlab/>

Chapter 1

Comparison Criteria

This Chapter defines the criteria that the evaluation of the approaches will be based on. For every criterion some of the SQFs it affects will be referenced. Different works have discussed software quality factors in software development [Barbacci et al., 1995] [Chung and Mylopoulos, 2000] and standards have been defined by ISO, IEEE and ANSI. Although there are similarities, they do not all agree on the same definitions and classification. This work will stick to the definitions as presented by [Chung and Mylopoulos, 2000]. Based on the criteria, a classification will be defined and the classes will be illustrated with an example of a suitable approach.

For some of the criteria such a classification is very hard to come up with. The approaches can be so different with respect to a certain criterion that one cannot identify the classes of approaches based on recurrent practices. Since having a class per approach is totally useless, we will resort to describing the essence of a technique, if such a technique is available in the first place.

Throughout this work and the examples, we will fall back on the concepts and techniques defined in the research papers that were studied. We do not include an introduction that explains these approaches or discusses these papers. It is up to the reader to study these approaches him- or herself.

The criteria defined in this chapter are introduced in the following (a more detailed schematic overview can be found in Appendix A.1). The two first criteria are related to aspect-oriented composition, the other criteria are related to the development process with special attention to design and its relation to other phases within the full life-cycle.

- *Compositional Separation*

The separation into symmetrically or asymmetrically organized paradigms for composition.

- *Compositional Expressiveness*

The type, complexity and power of the composition technique and its specification.

- *Alignment to Phases*

The alignment of a design approach to the requirements and/or implementation phase.

- *Refinement Mapping*

The refinement process can consist of (a) a single model that is iteratively refined, (b) a number of models, one for every step in the process, or (c) a hybrid approach.

- *Traceability of Design*

A dependency relationship (traceability) relates an artifact from one model to the derived artifact in another model, and vice versa. This relationship can hold across phase borders.

- *Process Support*

The support for a design process or list of guidelines that prescribe or assist the way a system is designed.

1.1 Definitions

1.1.1 Compositional Separation

Composition is a key concept of any paradigm in AO. An important distinction in the composition is the *symmetrical* vs. *asymmetrical* separation. In what follows the views and explanations of Harrison, et al. on this matter will be presented [Harrison et al., 2002]. When conducting the evaluation, some approaches got classified as symmetric, while we would intuitively classify these approaches as being asymmetric. Something seems to be missing in the definition as presented in the work of Harrison, et al. This will be discussed when we define the classification of “Element symmetry and Relation asymmetry”.

An **asymmetric** composition models aspects and components as separate entities. Aspects typically model crosscutting concerns, while components model the system’s “base”. These two model elements have distinctively different structures and play different roles in the composition. It is said that aspects are applied to (even “woven into”) the base components; and aspect-aspect, component-component compositions are generally not supported.

The **symmetric** composition on the other hand makes no distinction into aspects and components. It simply views the system as a set of concerns that are to be composed. It includes a means to separate all kinds of concerns, instead of separating the crosscutting concerns from the rest of the system. Choosing the right compositional separation for the situation can greatly affect the set of software quality factors that the design benefits from.

Symmetry and asymmetry are two extremes, most approaches are situated somewhere in between—they form a spectrum. Symmetric separation defines highly decoupled components. Though not all concerns can be separated that easily. There exists always some form of dependency between concerns that cannot be separated out. Even a symmetric approach should be able to allow some asymmetry. This is why it was decided to split this general concept of compositional separation into separation at the level of *element*, *relation* and

join point. In case of *element* we are dealing with the issue of structure: if there is a specific distinction between aspect and component or if a “base” model can be discerned, then we are dealing with element asymmetry. In case of *relation* we are dealing with *placement* on the one hand and with *scope* on the other. Placement is about the location of the composition specification. The specifications can be defined within some components, that specify how they are composed with other components —relation asymmetry. Or the specification may be separated out itself (not within either of the composed components), or placed anywhere —both relation symmetry. Relation scope is about the domain and codomain of the relation. If any aspect or component can be composed with any other aspect or component, then we have a symmetric scope. The scope is asymmetric if the relation only allows aspects to be composed with components, and not the other way around. Join point symmetry is a more low-level matter that is not really of use here, therefore it will not be discussed.

Classification of Approaches

The main classes that we identified by the classification into symmetric and asymmetric are discussed. Examples of the most important approach are employed in every category to try and make clear what differentiates them from each other. All examples are based on the observer pattern.

Element and Relation symmetry. The most important example in this class is Multi Dimensional Separation of Concerns (MDSOC). Hyper/J is an instance of MDSOC at the programming level. A few researchers have started addressing the problem at the design level.

Theme/UML [Clarke and Walker, 2001, Clarke, 2002, Baniassad and Clarke, 2004] is an example of such an approach (figure 1.1). System concerns, inferred from the requirements, are represented as design themes. Each concern is modelled into a theme, independent from any other concern. Themes are composed into an integrated design. It is important to notice that, aside from the pattern template parameters that need to be filled in, there is no discernible difference between what the authors call *template themes* and *base* themes. That should allow us to argue that we are dealing with symmetric elements, that are composed by a symmetrically scoped and placed relation.

- **Understandability.** Understanding a symmetric concern’s design when it is on its own is much easier because the concern is much more decoupled since it defines everything it requires (“declaratively complete”). An understanding of the base component is not necessary to understand the concern. When considering composition, it is more complicated to understand the whole, because of the added complexity of the composition specification. Apart from a binding it needs to deal with such issues as reconciling different class hierarchies and interfaces, especially when concerns are developed separately.
- **Evolvability.** Symmetric concerns are defined on their own, they depend very little on each other. This allows a concern to be evolved on its own

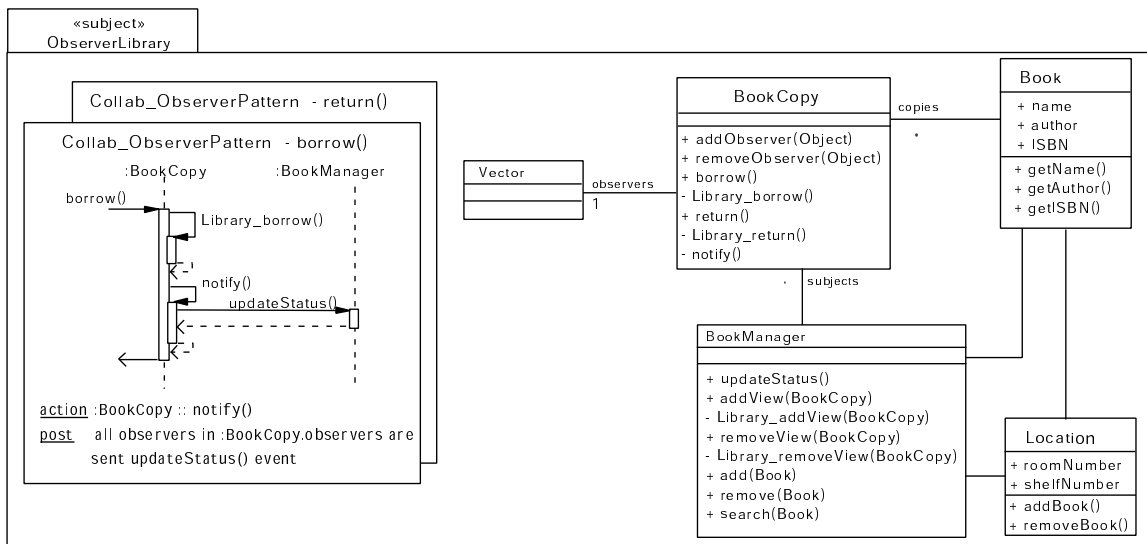
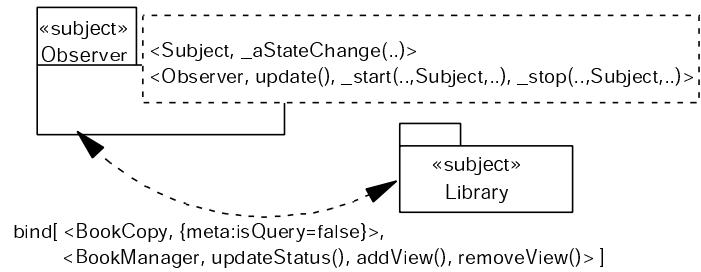


Figure 1.1: Theme/UML, (above) Pattern bindings (`bind[...]`) and `merge` integration (two-way arc) of `Observer` and `Library` themes, (below) `ObserverLibrary` theme as output of merge integration

without having to change any base components. The composition specification may also need to be evolved together with the concern. Because the specification is defined outside the concern it is potentially more accessible and easier to change.

- **Reusability.** Because symmetric concerns are highly decoupled and separated from their composition specification, it is clear that these concerns are generally much more suited to be reused in another context.

Element and Relation asymmetry. Typical examples of this class are approaches that try to model AspectJ programs. We discern a separation of elements into aspect and base types. The composition relation typically consists of a set of pointcuts, located inside the aspect, that binds the aspect to the base. An example of such an approach is AODM [Stein et al., 2002b, Stein et al., 2002a] (see figure 1.2 on the following page). The authors argue that a strong syntactical resemblance exists between standard UML concepts and those introduced by AOP: join point, pointcut, advice, introduction, aspect. *Stereotypes* are used to alter the semantics of these AOP concepts. We clearly see the asymmetrical separation into aspect and class, the one way relation from aspect to class (`<<crosscut>>`) and the definition of this relation inside the aspect (`<<pointcut>>`).

- **Understandability.** Aspects are defined as extensions to an existing base. This makes them more difficult to comprehend when considered separately, since they are highly coupled to the structure and behaviour of the base. Understanding the aspect requires an understanding of the base. In composition, this is actually the other way around. The aspect contains its relationship specification and is connected in a more straightforward way, without the need for reconciliation, making it less hard to see the whole.
- **Evolvability.** Since asymmetrical aspects are mostly extending existing behaviour, they are composed onto an existing structure. By design they are very dependant on this structure and any change to it may have an impact on the aspect. Evolving the base may invoke changes on all aspects depending on it. Evolving the aspect on the other hand has a much smaller impact, because it contains its own composition specification. A change can be incorporated simply by installing a new extension.
- **Reusability.** Aspects themselves state how they are to be composed to a base. This means the aspect is specifically designed to be applied to that base. Aspects generally cannot be extended by other aspects. These are all reasons why reuse in the asymmetrical case isn't that straightforward.

Element symmetry and Relation asymmetry. Approaches belonging to this category use the same structural elements but an asymmetric one-way relation between the two. The relation specification itself can be quite diverse: *AspectJ-like pointcuts*, *asymmetrical superimposition principle* and *aspectual collaboration connectors* (see section 1.1.2). This makes this category too differentiated to discuss its global effect on software quality factors. For this reason the discussion will be limited to the UFA approach.

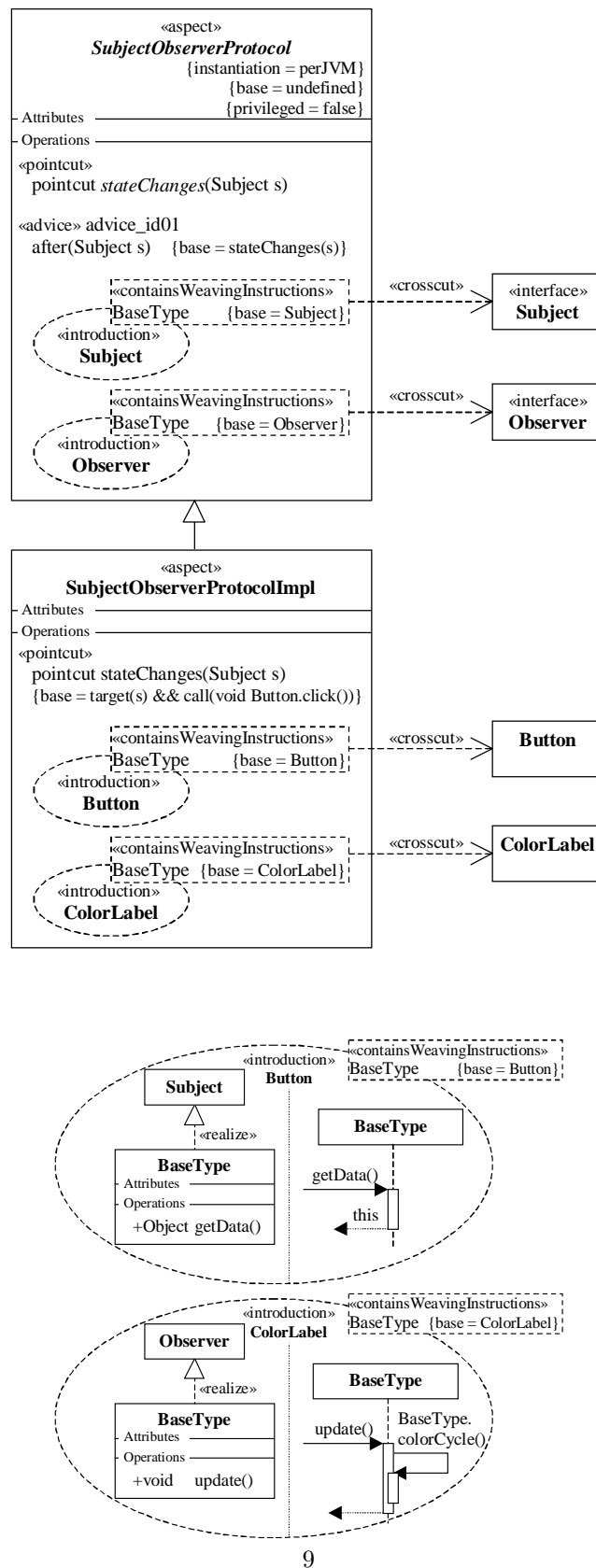


Figure 1.2: Aspect-Oriented Design Model (AODM), (above) Abstract Observer-pattern aspect refined by a concrete aspect applied to a specific application, (below) Design of introductions

UFA [Herrmann, 2002a] is an approach that illustrates this category (figure 1.3). The composition relation in this approach is a directed relationship that differentiates aspect from base, although aspects and base components are structurally similar elements. It is based on the aspectual collaboration principle [Lieberherr et al., 1999] in which a connector component represents the composition relation.

Approaches that use the *pointcut mechanism* are more related to the asymmetrical case, with the exception that the relation specification is separated out. Approaches that employ *superimposition* allow aspects to be applied to other aspects, but specifically address the inter-dependencies of these aspects and the order of appliance. This makes them lean somewhat to the symmetrical separation with the main difference that these approaches have increased coupling, which negatively affects SFQs like understandability, evolvability, reusability.

This brings us back to the issue of the Harrison, et al. definition of symmetry vs. asymmetry where it classifies some of the approaches differently from how we would. The issue lies within the concept of relationship symmetry, more specific its separation into scope and placement. The approach discussed here, UFA, satisfies both **scope** (**Observer** and **Library** both packages) and **placement** (within **ObserveLibrary** connector) **symmetry** (figure 1.3). So, according to Harrison, et al. the relationship should be classified as symmetric. However, in this case it is clear that this relationship is uni-directional, meaning that the **Observer** package is only allowed (or able) to extend the **Library** package. This conflicts with our understanding that symmetrical paradigms must allow a *merge* of two elements where both peers are able to take part in the relationship as equals and it is not one element that contributes to the other. The *directionality* of the relation seems to have an effect on the symmetry of the relation. A bi-directional relation can be said to have direction symmetry and a uni-directional one, direction asymmetry.

- **Understandability.** It is not immediately clear how the difference between the abstract aspects in the case of UFA and the template concerns in Theme/UML affect the quality factor of understandability, since they are both very alike. Although the difference between the composition relationship in UFA and Theme/UML is very apparent, we cannot determine how this exactly influences the understandability of the composition, since the actual composition of aspects and base is postponed until implementation.
- **Evolvability.** Evolvability of design in UFA is possible because the elements are decoupled through means of the connector that composes them. Both aspects and base (in a specific relation) can be evolved independently as long as the connector is able to compose them together.
- **Reusability.** Aspectual collaboration was introduced with increased reuse in mind. Aspects and base components are defined separately and connected afterwards. Aspects can be reused in another context, composed with another connector.

Element asymmetry and Relation symmetry. This classification was not found in any of the studied approaches. It doesn't make much sense to first define separate elements and then neglect the difference by composing them

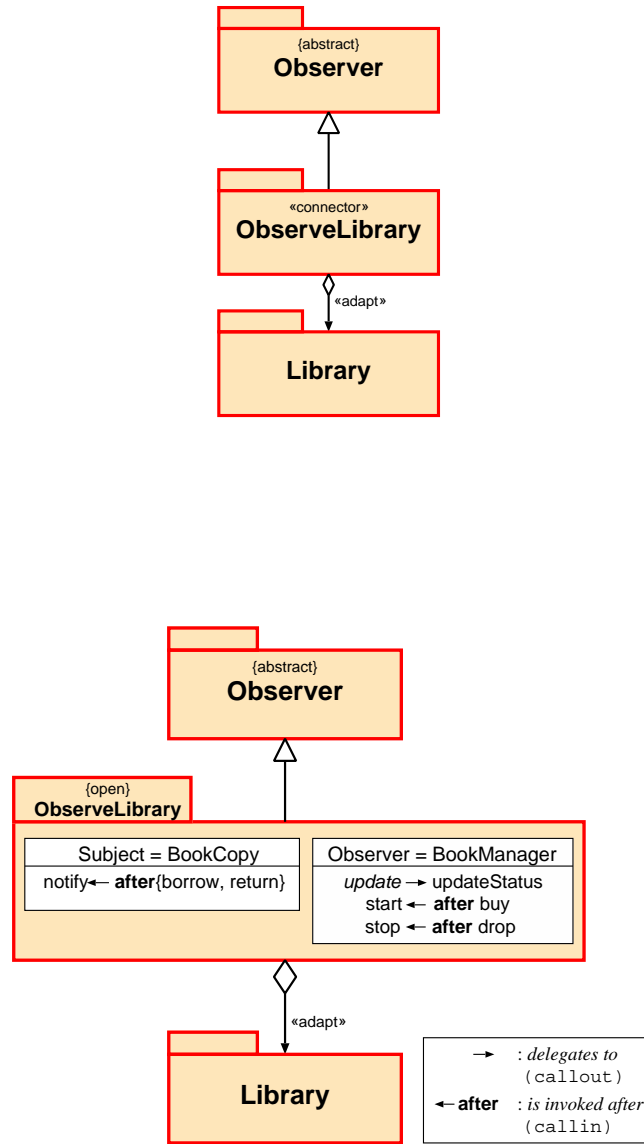


Figure 1.3: UML For Aspects (UFA), (above) Connector package *refines* Observer pattern and *adapts* Library core model, (below) Class and method *bindings* inside the connector package

using the same composition relation for every possible AO relationship between them.

1.1.2 Compositional Expressiveness

Composition has been discussed in section 1.1.1, when dealing with the criterion of Compositional Separation. Composition was called a key concept of any AO paradigm. The reason why it is so important follows from one of the goals of aspect-orientation. Separation of Concerns deals with the identification and modularization of concerns in software development at the conceptual and implementation levels. Development benefits in many ways from this separation [Hürsch and Lopes, 1995], but it is only truly useful if these concerns can be—at some point—reintegrated or recomposed into a system as a whole. AO approaches put forward new modularization units, like aspects, themes or hyper-slices. It is essential that these approaches also provide composition techniques to make this reintegration possible.

Composition requires a syntax: what a composition, the individual parts and the whole looks like. It also requires semantics: the meaning of composing certain parts, if it's even possible to compose and how to deal with conflicts. Separate development is one of the benefits of soc. Without close cooperation or strict agreements between parties, conflicts are impossible to avoid. Some approaches provide mechanisms for integration and reconciliation that can recover from these incompatibilities. What is also important is the abstraction level the composition is defined at. Composition can happen at the conceptual level or can be postponed until implementation. Composition should handle both the integration of structure and of behaviour. Some AO paradigms support the introduction of structural elements into advised models. This is a way of retaining encapsulation while extending the properties or behaviour of a model element. The addition or replacement of behaviour is common in AO approaches.

The Compositional Expressiveness criterion attempts to classify the approaches by the kind of composition mechanism they apply. It shows how some of these mechanisms are more elaborate than others. For example, introducing AspectJ's aspect, pointcut and advice concepts into a modelling language can enable design to at least represent AspectJ programs. However, this doesn't simplify reasoning about the composition at the conceptual level. The level of abstraction remains too low to be a means of understandability on top of the implementation. Hence, qualities like scalability, understandability and reusability do not score too high with these approaches. On the other hand, modelling languages like Theme/UML, AAM [France et al., 2004] or AVA [Katara and Katz, 2003] describe composition mechanisms at a much higher level of abstraction. These mechanisms are much more conceptual in nature. They are used to describe an actual composition at the modelling level, instead of reflecting some composition at the implementation level. Mind though, that this doesn't render the low abstraction approaches useless. Remembering what we said earlier about design, at some point the programmer that needs to implement the system will greatly appreciate these levels of detail.

Another concern of expressiveness is power vs. simplicity. Powerful composition specification can result in too much overhead. While a too simplistic specifica-

tion may harm the quality of the design. An approach that is used frequently is to define a default specification that applies to a general composition but that can be tweaked easily (like for example, matching classes by name [see Theme, AAM, ...]).

Classification of Approaches

It's difficult to define an exhaustive and exclusive list of composition mechanisms used by AOD approaches, since none of them agree on a single method. Therefore, a partial list of high level mechanisms, that reappear in different approaches under slightly altered form, will be given.

Binding, Integration & Reconciliation. Theme/UML (figure 1.1) or AAM are examples of this category. They define aspects as reusable templates that are *instantiated* by **binding** the parameters to concrete model elements. The next step is to **integrate** them into the model (by merge, override, ...) and then **reconciliating** the possible conflicts that arise.

- **Scalability.** This approach is very scalable. New concerns can be added incrementally and designed separately. The composition specification is defined at a high enough level of abstraction to scale well to larger system designs.
- **Understandability.** The level of abstraction the composition is defined at makes it also a lot easier to understand it, because one does not have to figure out complicated pointcut patterns.
- **Maintainability.** The fact that concerns or aspects are designed independently of each other and composed together by a powerful composition relationship makes that the impact of changes to other models is limited. This property is very useful when maintaining a design.
- **Evolvability.** Another factor that deals with change is evolvability. Where for maintainability the goal is to retain functionality with fewer errors, evolvability extends and adds functionality under changing requirements. It is clear that this factor also benefits from the limited impact of change property and the fact that concerns can be added incrementally.

Join Points and Pointcuts. Approaches like AODM (figure 1.2) include implementation-like expressive pointcuts into the model —sometimes accompanied by a <<crosscut>> relationship arrow. Depending on the generality of the pointcut this can severely clutter the model diagram.

- **Understandability.** Pointcuts are low-level constructs that appear to be very unintuitive for humans. It isn't immediately clear which elements in the design are affected when viewing a complex pointcut. Pointcuts can also take into account properties that are only available at runtime. This complicates things even more. It is clear that the pointcut technique is very powerful, but too low-level to be easily understandable for human developers. Many pointcuts may also clutter the design model, if they can be easily visualised in the first place.

- **Scalability.** Due to its complexity and clutter, pointcuts do not scale well to large system designs.
- **Evolvability.** Badly designed pointcuts may also result in unwanted compositions when the software is evolved. Newly added behaviour may accidentally trigger a pointcut, or existing behaviour may not trigger a certain pointcut as a consequence of some change.
- **Maintainability.** The same problems as for evolvability may occur when maintenance efforts change parts of the design.

Aspectual Collaboration Connectors. [Lieberherr et al., 1999] These approaches use a **connector** package, also called *adaptor*, to serve as a link between an abstract, reusable aspect and a base model element, that is said to be *adapted*. All *integration* and *reconciliation* logic is located inside the connector. The connector remains modularized from the moment it is defined during design, in the implementation and even at runtime. An example of an approach that employs this mechanism is UFA (figure 1.3).

- **Understandability, Scalability, Evolvability and Maintainability.** The connector component stores all composition information in a single location that is made into a first class model entity. This technique promotes the binding relationship to a first class entity that becomes a part of the design model and can even remain there through the implementation phase. Some approaches use high-level specifications, comparable to the specification used in the first category (*Binding, Integration & Reconciliation*), having a similar beneficial effect on the SQFs. Other approaches use pointcut-based techniques, that are now separated away from the aspect behaviour. This has some minor positive impact on the SQFs mentioned here. But the SQF affected most is that of **Reusability**. The model elements that are composed by this technique are completely freed from their composition specification, allowing them to be reused more easily. It can be seen as for example extracting the pointcut definitions from an AspectJ aspect and storing them in a location that represents the binding between the specific aspect and base.

An interesting observation concerning the three classifications above is that the first one describes a composition that is fully realized at design-time. The second class describes a composition that is realized at implementation-time. And last but not least, the third class describes a composition that is realized during design and continues on existing at implementation- and even run-time.

1.1.3 Alignment to Phases

The aspect-oriented software development process consists of different phases. Each of these phases is in some way related to the previous and next phase: the output of one phase is used as input for the next. Considering the entire process, design can be thought of as an intermediate phase, translating requirements to implementation. As such, a design notation should carefully *balance* its modelling support between the abstraction needed to address the output of

the requirement phase and the precision required to implement the given design. Assessing design notations by their abstraction over implementation is an incorrect way of interpreting the importance of abstraction at the design level. At some point the goal of design, and of software development in general, is to enable a developer to easily translate the design into some implementation technology. It seems these two requirements of design are in conflict with each other. But, isn't this what translation is all about? To function as an intermediate process between two conflicting languages. You can think of design as a translation process that uses stepwise *refinement* to move from an abstract to a detailed model.

When we look at a design model at a high level of abstraction, some software quality factors become very apparent: reusability, flexibility, understandability. Though at lower levels they seem to fade away completely. This is however not unexpected, since it's one of the main reasons software design, as it is regarded today, was introduced. Abstract away from implementation details, so we are able to comprehend, adapt and reuse the design more easily than working our way through lots of programming code.

An approach can be aligned to **requirements**, to **implementation**, provide concepts for **both** or be aligned to **none** of the phases.

Classification of Approaches

Aligned to Requirements. These approaches provide design concepts that map to a high-level concern structure while abstracting away from implementation-level design. These approaches do not clearly map to any implementation technology.

- **Understandability.** Higher levels of abstraction make the basic idea of the design more easily understandable. Although abstraction from detail is an important characteristic of design, good design should also capture this detail at some level in development.
- **Reusability.** Abstract design elements can be reused more easily in other designs. The lack of detail, trade-off and commitment to specific implementation technologies makes that the element being reused is much easier to adjust to the new context.
- **Evolvability.** Evolving a high-level design is easier to do because of the same reasons that enable reusability.

Aligned to Implementation. AODM is an example of an approach that is aligned to the implementation phase (figure 1.2). When first introduced, AODM was specifically designed to model AspectJ programs using an extension to UML. In [Stein et al., 2002a] this proposal was shown to be generalisable to other implementation technologies like Composition Filters, Adaptive Programming and Hyper/J [Ossher and Tarr, 2000]. Although for this last approach, it falls short in modelling the complete semantics of the composition relationship. This is because AODM is based on an augmentation model where aspects adapt a base (like AspectJ). These approaches have no high level modelling concept that maps the design to the concerns identified during requirements phase.

- **Understandability.** A design model at the implementation-level of abstraction is laden with details. It contains design trade-offs that have been made when committing to a certain implementation technology. All this has a negative effect on the understandability of the design model.
- **Reusability.** It is harder to reuse a design element in another context when it contains many details that connect it to its former context. For example, a class element that was designed to be implemented on the Java platform cannot be reused straight away in a design for the .NET platform. A high-level design pattern however can be reused in any OO implementation language and platform.
- **Evolvability.** Evolving software design at this level of detail is hard to do and leads to problems that are similar to those encountered when evolving programming code.

Aligned to Requirements and Implementation. An approach that supports this alignment is Theme/UML. It has a concept, called theme, that represents a concern defined at the requirement phase. The approach also allows detailed class and communication diagrams (see figure 1.1 on page 7, below). Theme/UML has been shown to map to implementation languages like AspectJ [Clarke and Walker, 2002, Baniassad and Clarke, 2004].

- The software quality factors discussions from the other classifications are both applicable here.

Aligned to None. There is no clear alignment of design to any of the other phases.

1.1.4 Refinement Mapping

Design can be regarded as a series of steps adding more and more detail to an initial abstract model. This way the design evolves from a higher level of abstraction to a lower level. When this step-wise refinement takes place inside the same model we call this “**extending**” the model. Using a single model has the disadvantage that we won’t be able to discern the individual steps directly from the model. The process should be well documented to retain understandability and trade-off information between steps.

A different approach is “**creating**” a new instance of the model for every step. These models would implicitly help documenting the evolution of the design. Some paradigms use a hybrid approach that defines different models but allows some internal single-model refinement.

Classification of Approaches

Extending UFA sees design as series of steps that add more and more details to an abstract model. It starts with a high level model that is refined during a number of iterations. The approach identifies aspect bindings (to the base) at a series of levels, going from package level, through class level, to method level.

- **Traceability.** When design is performed on a single model it requires an additional effort to maintain traceability between subsequent phases in development. When traceability isn't realized implicitly in the design model it can be added by documenting changes to the structure and trade-off information.
- **Understandability.** A single model doesn't show a record of the step-wise process that refines it. This can severely undermine the ability to understand the refinement that the model underwent (why decisions were made, how the structure evolved, the reason behind realized dependencies, etc.).

Creating CoCompose is an approach that refines its model by creating a sub-model for abstracted concepts in the parent model. At some point during design these models are “flattened” into a single model that represents the complete system design.

- **Traceability.** Design is a complex phase that turns small abstract models into large complex models that can be implemented in a straightforward way on a specific implementation technology. Creating traceability within design can be made easier by keeping a visual record of the refinement process in a series of related models. For example, CoCompose can realize a design pattern in a single model and abstract away the roles in this pattern by “*concepts*” that can be designed separately by other models. Besides implicitly keeping a record of the refinement process, this structure of related models also forms a nice basis to attach the relevant documentation to.
- **Understandability.** The understandability of the design process also benefits from this form of refinement. See the discussion of Traceability above.

Both The Theme/UML process consists of a number of steps and matching models that lead to a composed design (figure 1.4). Individual models (themes) can be separately refined.

- The software quality factors discussions from the other refinement mapping classifications are applicable here.

1.1.5 Traceability of Design

Traceability is about relating artifacts backwards and forwards in the development cycle to other representations of the artifact in different models and possibly across phases. Traceability is a property of the relation between models, where one is a refinement of the other. A property that indicates the clarity or transparency of the refinement process. More specific, from the moment an *artifact* is identified during requirements, traceability allows following it through the different representations it adopts during its life-cycle and enables the understanding of the decisions that affect it. It's about relating an artifact to its previous and next representations in an unbroken chain from requirements to

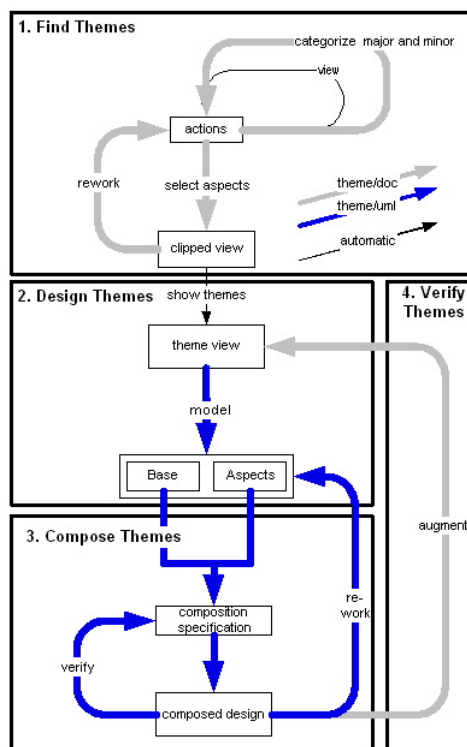


Figure 1.4: Theme Process

Legend		Classification	
R	Requirements	$D \rightarrow I$	design to implementation
A	Architecture	$A \rightarrow D$	architecture to design
D	Detailed Design	$R \rightarrow D$	requirements to design
I	Implementation	$R \rightarrow D \rightarrow I$	requirements to design and design to implementation
		$A \rightarrow D \rightarrow I$	architecture to design and design to implementation

Table 1.1: Classification overview of external traceability

implementation. This allows us to clearly see where an artifact comes from and where it goes to.

The concept of traceability¹ as it is defined here, is related to the **trace dependency** concept introduced in the UML [Rumbaugh et al., 2005]:

Trace Dependency. “An abstraction dependency that indicates a historical development process or other extra-model relationship between two elements that represent the same concept without specific rules for deriving one from the other. [...] It is intended to permit traceability of requirements across development.”

Traceability is mostly considered in the context of change. When altering an element of some model² (during requirements, architecture, design and even implementation), consistency requires this change to take place to the element’s representation in related models —models that are related by traceability.

We distinguish between two forms, namely *internal* and *external* traceability. The latter is about traceability in between the phases of the development process. It’s an indication of the transparency of the design between, on the one side, requirement models (or architectural models) and, on the other side, an implementation. An overview of the possible classification of external traceability encountered in the studied approaches can be seen in table 1.1.

Internal traceability deals with traceability of the models belonging to one phase in the life-cycle. Here, only traceability between the internal phases of design will be considered. Some of the design approaches we studied only describe a single model, in these cases there is no internal traceability. Formulating a classification for internal traceability is no easy task. The reason is that every AOD approach has a different way of handling the problem. For approaches that support some form of internal traceability the enabling key concept will be discussed.

Classification of Approaches

External Traceability “ $R \rightarrow D \rightarrow I$ ”. Theme fits into this category (figures 1.1, 1.4). It expresses concerns in requirement and design constructs called

¹builds upon traceability as defined in [Chitchyan et al., 2005]

²In the context of explaining traceability as a relation between models, we avoid using the term “artifact” since it also encompasses “model”.

themes. Themes are more general than aspects, they are also defined with a symmetrical separation in mind. At design level, Theme/UML's³ design level themes map perfectly to the themes identified at requirement level. Theme/UML is an extension of the UML, consequently it is traceable to an OO implementation. However, the theme packages that map to concerns are lost when the design models are composed. Theme provides some guidelines to implement themes on various AO implementation platforms [Clarke and Walker, 2001, Clarke and Walker, 2002].

Other classes of external traceability are shown in table 1.1.

- **Evolvability.** Approaches that are traceable to requirements phase have a design model that is based on the models created during this previous phase of development. Changes to these models may affect the design. For example when a certain stakeholder requirement changes. Traceability relates the artifacts in the models of previous phases to artifacts inside the design. This way it can help pinpoint the exact locations inside the design that are affected by these changes. However this doesn't ensure that every change will be trivial. Depending on the structure of the design in question some changes can be more invasive than others. Traceability is certainly not the key ingredient to evolvability in software development as it does not handle dependency relations, nevertheless it can prove to be very useful in tracking down artifact changes throughout the life-cycle.
- **Understandability.** Traceability is about adding structure and alignment to the models created during software development. This structure can serve as a frame for complementing the design with decision and trade-off information. Extended with tool support this can greatly assist in managing and understanding the software development process [Kowalczykiewicz and Weiss, 2002].

Internal Traceability. UFA uses packages to represent and encapsulate concerns (figure 1.3). Inside these packages are the diagrams that model structure and behaviour at the abstract, class and method levels. Abstract packages (stereotyped <<abstract>>) represent reusable parts of concern definitions. They are refined by a connector package (<<connector>>). A connector is composed with a base package by a directed adaption relationship. Inside the connector a mapping is defined at the class level: *Subject = BookCopy*, followed by a mapping at the method level: *notify* ← **after**{*borrow, return*}. This is an example of a technique that could allow some level of internal traceability.

None of the approaches we studied, agree on the same technique. Many of them do not even discuss the possibility of different stages inside design, making it difficult to identify a mechanism that allows traceability.

1.1.6 Process Support

Aspect-Oriented Design should be regarded as an integral part of a full life-cycle process. Aside from a modelling language —containing both syntax and

³The aspect-oriented design modelling language of Theme.

semantics— a design methodology, being a phase covering this life-cycle, should also describe a process. As an analogy, it takes more than knowing the vocabulary and being able to build correct sentences to turn language into a well structured text.

By process support we mean a well described, step-wise approach to modelling. This process must provide a road map to creating —preferably phased— design models that support the software quality factors discussed before. The design process, aside from the design language, has a large impact on the quality factors found in an approach.

It would be an incorrect assumption that every phased design process creates a model that bears the marks of these phases. A flat design model may have been build upon and refined at every step without ever showing what individual steps were added. This relates to the criterion of Refinement Mapping, which was discussed in section 1.1.4.

Some approaches do not provide an explicit process directing the design model refinement. Alternatively, an approach could supply the developer with guidelines that assist, when followed, in modelling the system. These guidelines can be very implicit; for example, supplying different kinds of diagrams allowing different viewpoints on the system being designed (structure, behaviour, state, ...) —not unlike UML. Allowing these diagrams to be linked together by a dependency relationship can benefit traceability. Some guidelines can be very explicit, like for example design patterns.

Classification of Approaches

Design Process. Theme/UML describes a process of identification and design of individual concerns, followed by an in-model composition (figure 1.4).

- **Understandability.** A design process can force a developer to bring structure in the way a system's design is built up. A step-wise process can split up design into a number of internal phases, showing more clearly how the design evolves from high levels of abstraction to lower levels. Annotated with refinement and trade-off information, this influences understandability even more in a favorable way. This phased design also enables **traceability**.
- **Maintainability.** A software system that is well designed, well documented and easy to understand can be better maintained. Good design applies techniques like encapsulation and low-coupling. It models and documents dependency relationships, builds a comprehensible model and anticipates the need for future maintenance. These qualities directly affect the maintainability of a system's design.

Design Guidelines. AAM [France et al., 2004] is an approach that provides guidelines on performing certain actions, like identifying concerns, developing composition strategies, evolving the design, etc. Concrete process support is shortly discussed in the open issues section of the referenced AAM paper.

- Guidelines have a limited effect on the way design is conducted in comparison to design process support. Although it can still have a beneficial effect on the software quality factors as discussed at the Design Process class.

Chapter 2

Approach Evaluation

This Chapter presents our evaluation of aspect-oriented design approaches using the criteria defined in Chapter 1. The basic idea is that for every criterion, the classification of all approaches is determined. A short discussion on the motivation for classifying an approach that way is included.

Determining the classification wasn't always straightforward. For some criterion we found several approaches that weren't easy to compare on the conceptual level—it would require a practical study to judge which performs better. For some other criteria the approaches were so alike that it wasn't clear where exactly to draw the line.

Another concern was that of the available information on the approaches. Some approaches are just at the early stages, propositions of interesting techniques. Others are worked out more into detail. When considering the state-of-the-art of these approaches, we had to be careful not to extrapolate on information that is not present in the research papers.

2.1 Classification Overview

The table on the following page gives a complete overview of the classifications for every criterion and every approach. The criteria form the columns, the approaches form the rows. As discussed in Chapter 1 some of the criteria have a subdivision that allows us to do a more detailed classification.

Approach	Compositional Separation		Compositional Expressiveness	Alignment	Refinement Mapping	Traceability of Design		Process Support
	Element	Relation				External	Internal	
AODM	asym	asym	aspectj based pointcuts	impl	n/a	D -> I	n/a	n/a
Theme	sym	sym	bindings, integration, reconciliation	both	creating / extending	R -> D -> I	abstract themes, themes, composed design	process
SUP	sym	asym	named transition/event matching	req	extending	R -> D	n/a	process
AAM	asym	asym	bindings, integration, reconciliation	none	creating / extending	A -> D	aspect models, context-specific aspects, composed model	guidelines
CoComipose	sym	sym	solution patterns	impl	creating	D -> I	composite nesting	n/a
UFA	sym	asym	connector, aspectual collaboration	impl	extending	A -> D -> I	package composition	n/a
AVA	sym	asym	concern diagrams and superimposition, dependencies	both	creating / extending	R -> D -> I	concern diagrams, (sub)aspects	n/a
AML	sym	asym	connector with aspectj composition	impl	n/a	D -> I	n/a	n/a
AOCE	asym	asym	required / provided aspect details	both	creating / extending	R -> D -> I	component aspects + guidelines	guidelines
UML4AOSD	asym	asym	expressive Pointcut relation	impl	n/a	D -> I	n/a	n/a
VC	asym	asym	n/a	impl	n/a	D -> I	n/a	n/a
ADM	asym	asym	pointcut based	none	creating	n/a	pure add/subt	n/a
DAOP/CAM	asym	asym	pointcuts on msgs, events; role based ID.	impl	creating / extending	A -> D -> I	mda-translation between models	process

2.2 Discussion

2.2.1 Compositional Separation

AODM was initially AspectJ based. It has a separation into *aspects* and *base* classes —element asymmetry. Pointcuts are defined as attributes of an aspect. This means AODM has an asymmetrically located composition specification. The scope of the composition relation is asymmetric because it deals with applying aspects to base classes. The relation direction is asymmetrical from aspects to base classes.

Theme/UML follows the symmetric approach to separation. Themes (called subjects in earlier work) are more general than aspects and base elements. A theme represents a concern and models both its structure as well as behaviour inside a UML package, independent from other themes —element symmetry. The composition specification is modelled as a relation between themes. Themes that are crosscutting are called aspect themes. These themes are templates that need to be instantiated by binding its parameters, allowing them to be applied in different contexts. The composition relation is symmetrically scoped and placed and allows a symmetric direction. Theme/UML is in some way related to what the MDSoc approach does at the level of implementation [Ossher and Tarr, 2000].

SUP uses statecharts to model aspect-oriented behaviour. Concurrent statecharts are used for modelling both normal and crosscutting concerns —element symmetry.

For the relationships between statecharts the event system is used. Events are the changes between the states identified in a concern. By providing names for these states, a certain statechart can trigger state changes in other charts. An implicit relation is thus created between state changes that send events and the ones that get triggered by these events. This relation has a symmetric scope, asymmetric placement within a statechart and an asymmetric direction—relation asymmetry.

AAM follows the asymmetric approach to separation. The approach describes *concerns solutions* —in essence requirements coupled with a goal— and concern solution models that describe how these are addressed in design. *Aspect models* in the approach describe crosscutting concern solutions in a generalised, context-free form. In other words: it is a pattern that characterises a family of logical (some high-level description) crosscutting concern solutions. The patterns are described using UML model templates, just like the Theme/UML approach does. The approach describes non-crosscutting concern solutions in a primary (base) model —element asymmetry. Aspect models are instantiated into an application-specific context before being composed with the primary (base) model —scope asymmetry. The bindings and directives are defined separately —symmetric placement. The relation has a symmetrical direction because both elements add to the composed model by being merged together. This approach thus has an asymmetric relation.

CoCompose uses *concepts* as the general modelling element. A concept is a generic element that allows developers to postpone the choice for a specific modelling (implementation) construct until the design is to be mapped on some implementation technology (aspect, class, interface, etc.) —symmetric elements. Composites are reusable abstractions of design concepts, such as dependency relations or design patterns or aspect-oriented mechanisms. A composite has roles that have to be filled in by concepts to instantiate the composite. A composite that contains other composites and concepts is called a *solution pattern* —symmetric relationship.

UFA uses more of a hybrid approach to concern separation. The approach models *aspects* as abstract packages, thus uses package incompleteness in favor of package templates —the latter is what Theme/UML or AAM do. Aspects adapt *core modules* (regular packages). Since both aspects and core use standard UML packages, we are dealing with element symmetry.

The relation between aspects and core is somewhat different. UFA uses a *directed* adaptation relationship together with a refinement relation. It realizes an integration by refining the aspect package while at the same time adapting a package from the core. This integration is represented by a *connector* package (figure 1.3) —whereas in Theme/UML a normal relation arrow is used. The relation has a symmetrical scope, symmetrical placement but an asymmetrical direction —relation asymmetry.

AVA proposes lifting up aspects to first class entities at design level. An aspect can be seen as an augmentation, mapping an existing design element into a new one with added detail. Essentially this means that elements defined in an OO decomposition that are relevant to certain aspects can be augmented with the information contained within those aspects —this is the authors' view. Let's try putting it another way —one that may sound more familiar— each aspect models its view on the design separately (guided by joint decision information), adding only those details to the elements that are relevant to the aspect (declaratively complete). Which makes us reach the conclusion that aspects are symmetrical entities.

Composition of aspects is based on the *superimposition principle*, an asymmetrical operation in which one aspect is applied on top of another —a directed relationship where one depends on, or adapts the other. As with UFA we classify the relation asymmetric. Scope and placement are symmetric but direction is asymmetric.

AML is related to the UFA approach in that it uses a connector package to link aspect packages to base packages. These aspect and base packages are the same kind of elements —element symmetry. But aspects do not support any form of abstraction or parametrization as UFA does. There are also no refinement and adaptation relations.

The approach employs a <<use>> dependency relation, but doesn't define its semantics. The connector contains all the AO information and is based on the low-level pointcut mechanism of AspectJ. The relation the connector represents

is symmetrically scoped and placed but is a directed relation —direction asymmetry, linking aspects to base packages —relation asymmetry.

AOCE identifies vertical slices of data and functionality (software components) —a typical object-oriented decomposition— and horizontal slices of functional and non-functional concerns (component aspects) —element asymmetry. Every aspect component has a number of aspect details describing the relationship to components and other aspects as required and provided entries in an interface-like description. These aspect details are separated in a textual specification language describing the component system —symmetrical placement. The approach is asymmetrically scoped, because there is a difference between requiring normal component services and aspect services and only component aspects provide aspect functionality. Interactions between interfaces are asymmetrically directed —asymmetric relation.

UML4AOSD defines aspect classes that extend regular classes (base classes) with aspect methods —element asymmetry. Aspect classes are linked by a directed association (a pointcut) to a class, thereby connecting aspect methods to some points belonging to the base class. These pointcuts are part of the model, represented by UML associations that are stereotyped with `<<pointcut>>` —symmetrically localised but asymmetrically scoped relationship with an asymmetric direction —asymmetric relation. The approach also introduces a unique —to our knowledge— structural element called *Groups*. A group can represent a number of heterogeneous or distributed design elements that form up a single concern space and can be linked to by a single pointcut.

VC defines *view components* to represent reusable functional concerns. This allows the concerns to be disconnected from the base model —element asymmetry. View components originate from views that perform a functional decomposition. They separate specific functional behaviour and data (attributes, operations, classes) into separate entities, while leaving the common behaviour and data in the base. These views are made into generic components by introducing the necessary missing elements. This allows to disconnect the view components from a certain base and reuse them by reconnecting to another base to form a design for a different system. This connection takes place by establishing a relation between the matching elements in a view component and those in the base —scope asymmetry. The model represents these relations as directed associations —direction asymmetry— between the view component’s elements and the base —placement symmetry. So it has an asymmetric relation.

ADM models concern behaviour using activity diagrams. The approach introduces a composition operation for activities that allows the “weaving” of *crosscutting concern activities* into a *primary activity model*. Aside from this semantical distinction in primary and concern activity, the latter one is extended by the *ActivityAddition* profile, that introduces two new types of activity nodes, the *InterfaceNode* and the *SubtractNode* —element asymmetry between primary and concern activity diagrams. Actually, every activity node is extended

by the profile, however, the nodes of the primary model are never used as an interface or subtractnode.

The actual composition of activities is defined by the two new node types introduced by the ActivityAddition profile. The primary activity contains *join/fork nodes* that come in pairs and are uniquely labelled. The interface or subtract nodes are essentially stereotyped *join/fork nodes* that contain attributes which state where these nodes are to be connected, by referring to the labels — asymmetrical placement. Concern activities can also contain labelled *join/fork nodes*. This allows the composition of concern activities with each other. The scope of this relation is asymmetrical since it only allows concern-concern and concern-primary compositions. The relation has a symmetric direction since both activities add to the composed activity. The relation is considered asymmetric.

CAM/DAOP describes a system’s design in terms of aspects, components and communication between them. These entities are disconnected from their final implementation or function by assigning them a unique role name (**ChatRole** for a **Chat** component, **PersistenceRole** for a **Persistence** aspect). Roles are an important part of the composition in the DAOP Component Aspect Model. Components and aspects are described by their respective interfaces. Components have *provided* and *required* services, while aspects have *evaluated* services instead of *provided* services —element asymmetry.

The *evaluated* interface includes the join points that an aspect is able to intercept and evaluate (execute corresponding advice). CAM/DAOP only allows interception of messages and events between entities’ interfaces. These join points describe pointcuts (not unlike AspectJ) that form the composition rules of CAM. The evaluate relation can be defined between aspect-component and aspect-aspect —scope asymmetry. The complete interfaces, including the composition, are defined separately of the entities (components and aspects) in an XML document —placement symmetry. The approach is considered asymmetric, although it allows aspect-aspect composition.

2.2.2 Compositional Expressiveness

AODM uses AspectJ pointcuts and composition semantics to compose aspects with base classes. However, there is no composition at design level. This means there is no model representing the composed system. Composition is completely postponed to implementation.

The approach uses join point designation and indication diagrams to separate the designation of join points (where) from the indication of the crosscutting aspects (what).

Theme/UML uses composition specifications that consist of bindings, integration relations and reconciliation directives. *Binding* instantiates template themes in a specific context, by binding the parameters to concrete theme elements. *Integration* is used to merge or override a specific theme with another theme. *Reconciliation* resolves compatibility problems by reconnecting

elements from different themes that represent the same concept or by disconnecting elements that match accidentally when using, for example, the “match by name” composition relation. This process turns separate themes into composite themes, that form the design of the system. Semantics of the integration are defined.

SUP is based on matching the names of state change events between concurrent statecharts. Composition uses events between objects, which increases coupling, or broadcasting events, reducing coupling but increasing overhead. Composition semantics are not defined.

AAM supports a composition mechanism that is very similar to Theme/UML composition. The *binding* process is very much alike in both approaches, instantiating templates to a specific context. Integration of aspect and primary models is guided and reconciliated by composition directives. Semantics of the integration are defined.

CoCompose uses composite solution patterns that represent a high-level structure of a design concept (for example the observer pattern applied in a system). Composite solution patterns consist of other composites, concepts and roles. Roles are essentially placeholders that need to be wired to concepts. A composite is a reusable abstraction of a design concept (relation, pattern, AO mechanisms, etc.). A concept is the generic main language element that present some design construct (aspect, class, method). The approach has no well defined composition semantics and is not based on the UML.

UFA is based on the aspectual collaboration model, which allows separate development of modules and a posteriori integration. The integration is realized by a refinement and adaptation relationship. A connector package refines an abstract aspect package to initialize an application specific aspect. Next, the connector package adapts a core package. The composition relation—consisting of a refinement and adaptation relation—is a directed relation, this way the adapted package (the core) is unaware of the adaptor package (the connector).

AVA uses a composition mechanism between aspects that is comparable to the of Theme/UML. Aspects consist of a “*uses*” part and a “*defines*” part. The uses part is said to be the pointcut that relates aspects to other aspects. But actually it is more than just a pointcut, it defines elements that the aspect needs and by doing this it actually states the structural and behavioural assumptions about the underlying system. These assumptions may be fulfilled by the aspect itself, or it may be specified that some other aspect is responsible for doing this. In essence this is very much alike to the merge and override integration of Theme/UML where themes complete each other when they are composed. The aspects that need to be composed to fulfill a certain concern are shown on the concern diagrams and implicitly by using the same name for matching aspect elements (in the uses-part). The actual composition technique that is used between two aspects that are to be composed is that of *superimposition*.

Superimposition is a directed relationship that differs from the typical bidirectional merge operation used in Theme/UML or MDSoc. In this way, the relation communicates the *order* in which aspects are to be composed at programming and run-time-level. The authors believe strongly that “*the concerns at the design level can and should be reflected in the programs, by using programming level constructs for aspects. Otherwise, most of the power of aspect-orientation is lost.*”

AML composition is realized by a connector package that contains the pointcut, advice-type and introduction knowledge. This way the aspect behaviour (and possible structure) is linked to the base. Base and aspect have no direct relation. The composition is deferred until implementation.

AOCE depends highly on component interfaces that describe the composition relations between components and component aspects. These interfaces consist of *provided* and *required* entries that represent normal component services together with aspect functionality. *Required* aspect details are the join points of the composition. These required details are fulfilled by a matching *provided* aspect detail. Actual composition is postponed until run-time and is visualized during design by means of collaboration diagrams. The actual composition semantics are not well defined.

UML4AOSD uses directed pointcut associations to link aspect classes to base classes or groups. These pointcuts can be quite expressive as they can be annotated with *role* and *cardinality* parameters. Role parameters determine which aspect method is applied at which base program points (described by a detailed expression). Cardinalities are used to relate the number of instantiated aspects to the number of instantiated base classes. The actual composition is postponed until runtime.

VC supports composition by relating elements from the view components to matching elements within the base, one by one. The relation is represented by a directed association. This leads to a great number of dependencies between the components, harming scalability. The approach also introduces a number of meta-classes (**ViewClass**, **ViewAttribute**, etc.) to the UML that require their own set of design and connection constraints that further complicate the use. The approach does not define semantics for the composition, nor a way to represent the model in its composed form.

ADM uses a composition mechanism that is comparable to pointcuts. The primary model defines labelled *join/fork nodes* that can be referenced to by an attribute of the stereotyped *join/fork nodes* defined in the activity concerns models. Depending on the actual stereotype being an interface node or a subtraction node, the concern activity will be added or removed from the primary model. Concern activity models can also include labelled *join/fork nodes*, allowing additive modification by other concerns themselves. The attribute contained

in the interface or subtraction node is essentially the pointcut that references a join point, being a labelled node.

CAM/DAOP supports the definition of join points as part of the evaluated interface of an aspect. Possible join points are the interception of sent or received messages and events between entities (aspects and components). The approach models entities as black-box elements, disallowing the interception of behaviour internal to a component. Other events that can be captured include the creation and destruction of component instances. The composition information is described separately in an XML document, together with other component and aspect related information. The approach provides in loosely coupled communication—the mechanism used for composition—by assigning unique Roles to all entities. Entities are further decoupled by storing any type of shared information inside uniquely named properties. These properties explicitly model dependencies that exist between entities. Properties are particularly useful as a way to synchronize interaction between non-orthogonal aspects.

2.2.3 Alignment to Phases

AODM is shown to abstract away from a specific implementation technology [Stein et al., 2002a]. Still, the approach uses low-level AOP concepts like aspect class, advice method, pointcut. There are no concepts representing concerns at the requirement level, nor are there any high-level (architecture level) concepts that abstract away from implementation details. This leads us to conclude that the approach is aligned to *implementation*.

Theme/UML uses theme elements that map to concerns identified during *requirements*. Every theme can be separately refined to a low-level detailed design. After integration, it can be *implemented* on a specific platform [Clarke and Walker, 2002]. This approach is aligned to *both* phases.

SUP is a high-level design approach, because it represents concerns and cross-cutting concerns in state charts [Aldawud et al., 2002]. The obvious question is whether this approach is still comprehensible on more complicated systems, with many more states and transitions. At lower levels the state transitions seem to conform to abstract transition events. The example described in the paper uses method-level calls but one can wonder if these are still possible in more complex application designs. The approach is aligned to *requirements*.

AAM uses aspect models and a primary model that describe solutions to concerns. These concerns solutions are represented at the architecture level. This is because the authors stress the goal-related nature of concern solutions. However they originated from concerns that were identified during *requirements* analysis. The approach abstracts away from detailed implementation-level design. The approach is not aligned to any phase.

CoCompose can be factored or transformed into different *implementation* technologies. The approach also supports a high-level design overview but no constructs to map to the concerns identified during requirements. The approach is aligned to *implementation*.

UFA starts from an architecture-level description of the system, where aspects and core are identified. Aspects are refined through a connector package at the class and method levels. UFA was created to model aspectual collaborations for the LAC programming language [Herrmann and Mezini, 2001] and its successor Object Teams [Herrmann, 2002b]. The approach is aligned to *implementation*.

AVA provides concern diagrams that model the main features and the aspects that collaborate to realize these features. This high-level overview is rather aligned to *requirements*. AVA models aspects as UML packages and allows them to contain all kinds of UML diagram types. At the lower levels of abstraction, this allows the approach to provide detailed design diagrams that can be mapped to an *implementation*.

AML supports low-level detailed design models that can be used to automatically generate AspectJ implementation skeletons. The approach provides no high-level concepts to model concerns. It is clearly aligned to *implementation*.

AOCE is defined as a process from requirements to implementation. Design builds further on the concerns and features defined at the requirements phase. It provides design level aspect details that are direct refinements from those at requirements level. The approach is created to design systems for a specific framework. Design can be used to create an *implementation* in that framework. The approach is aligned at *both* phases.

UML4AOSD does not provide any high-level elements mapping to requirements concerns. The approach was created to represent an AO design with a relevant amount of detail so that it should be possible to implement an automatic mapping from the design to concrete implementations in AspectJ-like languages. The approach is aligned to *implementation*.

VC derives a functional decomposition into view components from standard OO design. These components do not map to concerns identified during a requirements analysis. Consequently, the approach is not aligned to requirements. On the other hand, the approach has been shown to map to *implementation* platforms like CORBA, Fractal and EJB.

ADM uses the term crosscutting concerns, but does not clearly define where these concerns originated from. The level of abstraction is also not high enough to be directly mapped onto requirements-level concerns. At lower levels it remains unclear how the activity diagrams can be directly mapped onto implementation concepts. The approach is not aligned to any phase.

CAM/DAOP has no high-level structure that maps directly to requirements. However the computational model (C-M) that DAOP uses in its model-driven-architecture process, allows adding constraints expressing non-functional requirements, in the form of a dependency relation. The Component Aspect Model translates fluently, through a number of other models, into an *implementation* for the DAO-platform.

2.2.4 Refinement Mapping

AODM represents a low-level design model of implementation based AO approaches. Since there are no higher-level design concepts that need to be refined, refinement mapping is not applicable to this approach.

Theme/UML supports the level of abstraction required to align to the requirement phase, as well as the implementation phase. To integrate these levels of abstraction into a single design it needs a mapping between the different levels of abstraction. Theme/UML allows every concern to be modelled and refined separately. At some point all these models are composed into a new model that represents the complete system. In this way the approach is both extending and creating with respect to the use of design models.

SUP uses a design method that starts by defining high-level concerns as concurrent statecharts that are individually refined with states and events. Statecharts are not used to model low-level implementation events or method calls. The transitions between states are abstractions that represent a sequence of underlying behaviour.

AAM uses an approach that resembles that of Theme. Concern solution models are individually designed and refined by class diagrams and OCL definitions. These models are eventually integrated with a primary model into an integrated design model, steered by composition directives.

CoCompose does not have strictly defined levels of refinement. Though, one can define a composite containing other composites. These sub-composites can be presented as single element abstractions, without going into a refinement at that level. They can be refined separately on a different diagram. This method forms a hierarchical model nesting that is also a form of mapping.

UFA focuses on defining bindings on multiple levels between aspects and base. The approach describes methods for composition on package, class and method level. The approach uses a level-wise refinement of its module elements.

AVA identifies the main features and their dependencies in concern diagrams. Each concern consists of a collection of aspects that are separately developed. Some of these concerns overlap as a consequence of sharing an aspect. Aspects may prove to contain sub-aspects that are interesting in their own right, but weren't identified initially. The reverse is also possible, when it is more interesting to consider a number of aspects as a whole. Eventually all these aspects will be composed into a composite design model. Note however that this composition relation is completely different from the aspect-sub-aspect relationship.

AML does not have any internal design levels since it only describes a single low-level implementation model. This means there is no refinement of higher level concepts. Refinement mapping is not applicable to this approach.

AOCE provides an creating and extending refinement process that refines the aspect details identified at requirements. It uses different kinds of representations during this refinement. This refinement is described by guidelines.

UML4AOSD represents a low-level design model of implementation based AO approaches. Since there are no higher-level design concepts that need to be refined, refinement mapping is not applicable to this approach.

VC does not support internal levels of design. The approach defines a way to represent behaviour and structure belonging to a single concern separately in view components. There is no reference to a method for creating such a decomposition.

ADM uses model transformations to combine different activities models together in a single composed model. This allows to refine models with cross-cutting concerns in an additive and non-intrusive way, clearly isolating added behaviour from the initial model. This allows simple visualization, change or removal of the modification.

CAM/DAOP employs an MDA-like approach that translates a computational viewpoint model (C-M) of the high-level architecture into a component aspect model (CAM) which is again translated into a DAOP specific model that can be implemented (or possibly automatically translated) in the platform. Every step within this MDA-like translation process can be further refined by marking the elements in the model with meta-information relevant for the transformation to the next step.

2.2.5 Traceability of Design

AODM provides external traceability from low-level design models to different implementation technologies. Without any internal phases of design, there is no internal traceability.

Theme/UML supports external traceability from requirements to implementation. Themes/UML's themes map directly to the concerns identified during the requirements phase. These themes are represented by UML packages that are internally refined by several kinds of UML diagrams (class diagram, sequence diagram, ...). Using well defined composition semantics, these themes are combined into composite themes. Together, these themes form the refined design of the system that can be implemented. Internal traceability follows from the internal refinement of separate themes and the well defined composition semantics that translate separately refined themes into composite themes.

SUP supports external traceability from requirements to design. High-level concurrent statecharts map to concerns identified during requirement analysis. There is a very limited possibility of external traceability to the implementation phase, since defining states and events at the required low level of detail is unrealistic in complex situations. There is no internal traceability since the approach does not have any internal phases of design.

AAM supports external traceability from high-level architecture to detailed design. There is no direct traceability to an implementation technology. Internal traceability is supported between aspect models that instantiated into context-specific aspects models and later on composed with a primary model into a composed design model.

CoCompose supports external traceability from design to implementation. Concerns and composites can contain different implementations. These are used by the approach to automatically translate the design into an implementation. Since the approach has no well defined refinement phases the internal design, refinement structure is rather ad-hoc. Although this could allow some form of internal traceability.

UFA supports external traceability from high-level architectural design to implementation. Internal traceability is supported by well-defined aspect refinement at package, class and method level.

AVA provides external traceability from concern identification at requirements to implementation. Concerns are modelled to contain aspects and are allowed to depend on each other. The approach can be mapped onto certain implementation technologies that support the superimposition mechanism. Internal traceability is made possible between the concerns diagrams, aspect designs and composed design model. Aspects and sub-aspects also form a traceable relation.

AML provides external traceability from design to implementation because it allows an immediate traceability translation from design structure to implementation structure. The approach has no internal phases, consequently there is no possibility for internal traceability.

AOCE has a design phase that is steered by guidelines that take requirements and make them into an application. These guidelines enable some kind of internal traceability, as well as external traceability from requirements to design and design to implementation. The concept of component aspect is used throughout requirements, design, implementation and deployment.

UML4AOSD provides some level of external traceability from design to implementation, that should allow an automated mapping from design to AspectJ-like languages. Since there are no internal phases or refinement guidelines within design, there is no support for internal traceability.

VC can be translated to the implementation level, but it is unclear if this is a traceable relation. The approach does not support internal traceability because it has no internal phases of design.

ADM supports internal traceability. The approach employs model transformations and an additive operation for composing activity diagram models. Using pure net addition and net subtraction separately instead of general net addition, allows more readable and traceable model modifications. The approach doesn't support external traceability since it isn't aligned to any other life-cycle phase.

CAM/DAOP supports external traceability from high-level architecture view through CAM architecture design to implementation in the DAO-Platform. Internal traceability is supported by the MDA-like translation approach that uses a number of design phase models ranging from high-level architecture views to platform dependant design models.

2.2.6 Process Support

AODM provides no process description or guidelines that allow a developer to infer the design model from previous phases in the development process.

Theme/UML defines a process for designing a system that starts from requirements, up to an implementation. Themes are inferred from concerns identified during requirements analysis. These themes are independently designed using OO-methodologies and later on composed into a single design by composition relations. Theme defines a series of guidelines for implementing the design on certain platforms.

SUP describes a process, starting from requirements, to modelling the state-charts at the level of design.

AAM discusses high-level guidelines to process support that describe an iterative and incremental modelling process.

CoCompose defines no process or guidelines to conduct the system design. Translation to implementation, however, is described as an automated process.

UFA does not define any process or guidelines for using the design language. Although it acknowledges the importance of seamless integration of AO in the software life-cycle.

AVA does not define any process support or guidelines on how to use the modelling language to design a system. They mention however that aspects are identified during iterative development and that invested effort in architecture should pay off during maintenance.

AML has no definition of a process or guidelines to creating a design model.

AOCE describes certain guidelines for applying the approach to requirements, design and implementation phases and relating the phases to each other.

UML4AOSD does not define any process or guidelines for creating a design model. Although it does foresee some automated mechanism for mapping design to implementation.

VC does not define a process or guidelines for creating a design model.

ADM has no definition of a process or guidelines for creating the activity models.

CAM/DAOP describes a MDA-like process to refining the design model.

Conclusion

In this work we carried out a study and evaluation of aspect-oriented design approaches from the viewpoints of full life-cycle support for aspect-orientation and product-line engineering of large-scale systems. Evaluation criteria like *Alignment to Phases*, *Traceability of Design* and *Process Support* are an indication of the importance of the full life-cycle viewpoint.

Product-line engineering concerns the development of systems out of reusable building blocks. The criteria of *Compositional Separation* and *Expressiveness* refer to these building blocks and how they are composed. The concept of composition, that these two criteria characterize and try to classify, is very important to the field of aspect-orientation —as it is considered the dual of separation of concerns.

The study is based on literature of state-of-the-art approaches in aspect-oriented design. This initial work has its limitations, for example, it doesn't perform a case study to evaluate and compare approaches —such a case study may better capture the differences between the approaches. Instead we base our evaluation on six criteria, inspired by [Chitchyan et al., 2005], that are related to the goals of this work. We describe these criteria in Chapter 1 and discuss the classification that was determined for each criterion, together with the influence each class may have on some important software quality factors of the designed system.

The following paragraphs will discuss some observations we have made while classifying and evaluating the approaches in this work.

Limited support for a design process. An important observation, in the context of full life-cycle, when looking at the process support column of the classification overview table on page 24, is that many of the approaches do not support a clear process for conducting design. These approaches focus more on a modelling language to visualize concerns during design and less on actually placing this modelling language in the context of a development process. Although unfortunate for the goals of this study, this is understandable when considering that most of the approaches are still in a very early stage, as the research domain of AO modelling is only a few years old. However, the process support column also shows that some approaches did evolve beyond language support. They define an internal design process (eg: DAOP) or are even part of an external process or guided development that covers more than one phase of development (eg: Theme, AOCE). These approaches are built upon frameworks or platforms and acknowledge the need for a systematic way to develop software.

They represent an important step forward on the road to full life-cycle support for software engineering.

Classification themes within Compositional Expressiveness. Composition of concerns is an important part of aspect-orientation. It is the dual of separation of concerns, which is what AO is all about. The way in which concerns are composed together has an immense influence on the software quality factors an approach supports (scalability, understandability, reusability, evolvability, maintainability, etc.). It determines when the actual composition takes place (design, implementation, deployment, run-time), where its specification is defined and what it looks like. Compositional Expressiveness is possibly the criterion that best characterises an approach.

Reoccurring composition themes we encountered in slightly varying forms is the one used by Theme/UML and AAM. There, template aspects are firstly bound to a context and later integrated and reconciliated into another design model. Another popular approach is one that is based on AspectJ-concepts. This approach models join points together with the pointcuts that reference them. This is a low-level approach to modelling that lays close to the implementation level. This has its negative effects on various SQFs in case an approach exclusively supports this level of design. Another approach to modelling is that of aspectual collaboration connectors. This approach raises the composition relation to a first class design element (eg: packages, components). This element stores all the information relevant to the composition of the elements that are related by the connector.

It should be clear that there are many different kinds of composition techniques. *Every technique has its advantages as well as disadvantages. Choosing an appropriate technique is highly dependent on the context it will be used in.*

The connector technique, for example, is very popular in component based approaches, as the connector itself can be represented as a component. This makes the connector component interchangeable by another slightly different connector.

The pointcut composition approach are very useful when translating a design to a particular aspect-oriented approach that employs pointcuts. High-level design defines a composition at a more understandable and maintainable abstract level that can be refined to a detailed pointcut composition specification. Just to stress again that design isn't about abstraction alone.

Symmetrically vs. asymmetrically separated paradigms. While evaluating the approaches, several of them (UFA, AVA, AML) were particularly hard to characterize by the definition of symmetrical or asymmetrical paradigms as it is given in the work of Harrison, et al. [Harrison et al., 2002]. With this definition these approaches are clearly characterized as being symmetrical in *element* as well as in *relation*, however our understanding of symmetry vs. asymmetry—where elements should be able to take part in the relationship as equals—would make us characterize the composition relation as being asymmetrical because of its directional nature. While these approaches have some symmetrical properties, their relationship is an asymmetric one (see discussion at 1.1.1 on page 10).

An important observation is the tendency towards the asymmetrical paradigm. Many of the approaches are based on the composition model where an aspect

extends the base design. This is probably a direct consequence of the popularity of this technique in the field of AOP and the fact that these approaches are mapped to these kind of implementation techniques.

Dependence on the UML. Tables A.1 and A.2 on page 45 clearly show that the majority of approaches are in some way related to the UML. The approaches extend the UML in order to support the modelling of concerns or aspects and their composition with other concerns or base elements. This is because the UML is a very popular modelling language that is easily extensible with new syntax and semantics. Many modelling tools have been developed that support the UML, as have many OO software engineering approaches. Aspect-orientation is also an extension to object-oriented. *It seems to be sensible to model AO by extending the UML.*

Overall alignment to implementation. Many of the approaches we studied have the tendency to be aligned to the implementation phase, that is, they can be mapped with relative ease to an AO implementation technique. Much less approaches map to the requirements phase in the same way. This is probably because aspect-orientation originated in the programming world [Kiczales et al., 1997]. The implementation-aligned approaches build on existing AOP work that has proved to be successful, like AspectJ.

Appendix A

Overviews

A.1 Criteria

This section presents an overview of the comparison criteria defined in Chapter 1, as well as their possible subdivisions, classification and impact on SQF.

Compositional Separation

- subdivision into *element* and *relation*
- classification
 - symmetric
 - asymmetric
- impact on SQF (based on [Harrison et al., 2002])
 - evolvability
 - understandability
 - reusability

Compositional Expressiveness

- classification
 - pointcut based
 - bindings, integration, reconciliation
 - aspectual collaboration connector
 - name matching
- impact on SQF
 - scalability
 - understandability
 - maintainability
 - evolvability

Alignment to Requirements and Implementation

- classification
 - requirements
 - implementation
 - both
 - none
- impact on SQF
 - understandability
 - reusability
 - evolvability / flexibility

Refinement Mapping

- classification
 - creating
 - extending
 - both
 - n/a
- impact on SQF
 - traceability
 - understandability

Traceability of Design

- subdivision into *internal* and *external*
- classification
 - internal:
enabling key concept
 - external:
subset of $R \rightarrow A \rightarrow D \rightarrow I$
- impact on SQF
 - evolvability
 - understandability

Process Support

- classification
 - process
 - guidelines

- none
- impact on SQF
 - understandability
 - maintainability
 - traceability

A.2 Approaches

1. **AAM**: Aspect-oriented Architecture Models
[France et al., 2004]
2. **ADM**: Activity Diagram Modelling
[Barros and Gomes, 2003]
3. **AML**: Aspect Modelling Language
[Groher and Baumgarth, 2004]
4. **AOCE**: Aspect Oriented Component Engineering
[Grundy, 2000]
5. **AODM**: Aspect-Oriented Design Modelling
[Stein et al., 2002a]
[Stein et al., 2002b]
[Stein, 2002]
6. **AVA**: Architectural Views of Aspects
[Katara and Katz, 2003]
7. **CAM/DAOP**: Component Aspect Model / Dynamic component and Aspect-Oriented Platform
[Pinto et al., 2003]
[Fuentes et al., 2003]
[Pinto et al., 2005]
8. **CoCompose**
[Wagelaar, 2002]
9. **SUP**: State charts UML profile
[Aldawud et al., 2002]
[Aldawud et al., 2003]
10. **Theme/UML**
[Clarke and Walker, 2001]
[Clarke and Walker, 2002]
[Clarke, 2002]
[Baniassad and Clarke, 2004]
11. **UFA**: UML for Aspects
[Herrmann, 2002a]
12. **UML4AOSD**: UML for Aspect-Oriented Software Development
[Pawlak et al., 2002]
13. **VC**: View Components
[Muller, 2004]

A.3 AOD and UML

Many of the approaches we studied extend the UML. These tables list the kind of diagrams every approach employs and possibly extends with AO concepts.

	Class	Package	Component	Use Case
AODM	x			
Theme/UML	x	x		
SUP	x			
AAM	x	x		
CoCompose	not UML based			
UFA	x	x		
AVA	x	x		
AML	x	x		
AOCE	x		x	
UML4AOSD	x			
VC	x		x	
ADM				
CAM/DAOP	x		x	

Table A.1: Structure Modelling using UML diagrams

	State	Use Case	Activity	Sequence	Communication
AODM		x			x
Theme/UML				x	
SUP	x				
AAM					x
CoCompose	not UML based				
UFA					
AVA	x			x	
AML					
AOCE					x
UML4AOSD					
VC					
ADM			x		
CAM/DAOP					

Table A.2: Behaviour Modelling using UML diagrams

Bibliography

- [Aldawud et al., 2002] Aldawud, O., Bader, A., and Elrad, T. (2002). Weaving with statecharts. In *Proceedings of the Aspect-Oriented Modeling with UML workshop (at AOSD)*.
- [Aldawud et al., 2003] Aldawud, O., Elrad, T., and Bader, A. (2003). UML profile for aspect-oriented software development. In *Proceedings of the Aspect Oriented Modelling workshop (at AOSD)*.
- [Baniassad and Clarke, 2004] Baniassad, E. and Clarke, S. (2004). Theme: An approach for aspect-oriented analysis and design. In *Proceedings of the International Conference on Software Engineering*.
- [Barbacci et al., 1995] Barbacci, M., Longstaf, T. H., Klein, M. H., and Weinstock, C. B. (1995). Quality attributes. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh Pennsylvania 15213 USA.
- [Barros and Gomes, 2003] Barros, J. P. and Gomes, L. (2003). Towards the support for crosscutting concerns in activity diagrams: A graphical approach.
- [Chitchyan et al., 2005] Chitchyan, R., Rashid, A., Sawyer, P., Bakker, J., Alarcón, M. P., Garcia, A., Tekinerdogan, B., Clarke, S., and Jackson, A. (2005). Survey of aspect-oriented analysis and design. AOSD-Europe Project Deliverable No: AOSD-Europe-ULANC-9. Editor(s): R. Chitchyan, A. Rashid.
- [Chung and Mylopoulos, 2000] Chung, L., N. B. Y. E. and Mylopoulos, J. (2000). *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers.
- [Clarke, 2002] Clarke, S. (2002). Extending standard uml with model composition semantics. *Science of Computer Programming*, 44(1):71–100.
- [Clarke and Walker, 2001] Clarke, S. and Walker, R. J. (2001). Composition patterns: an approach to designing reusable aspects. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 5–14, Washington, DC, USA. IEEE Computer Society.
- [Clarke and Walker, 2002] Clarke, S. and Walker, R. J. (2002). Towards a standard design language for aosd. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 113–119, New York, NY, USA. ACM Press.

BIBLIOGRAPHY

- [France et al., 2004] France, R. B., Ray, I., Georg, G., and Ghosh, S. (2004). Aspect-oriented approach to early design modelling. *IEE Proceedings - Software*, 151(4):173–186.
- [Fuentes et al., 2003] Fuentes, L., Pinto, M., and Vallecillo, A. (2003). How mda can help designing component- and aspect-based applications. In *EDOC '03: Proceedings of the 7th International Conference on Enterprise Distributed Object Computing*, page 124, Washington, DC, USA. IEEE Computer Society.
- [Groher and Baumgarth, 2004] Groher, I. and Baumgarth, T. (March 2004). Aspect-orientation from design to code. In *Proceedings of the Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design; AOSD 2004*.
- [Grundy, 2000] Grundy, J. (2000). Multi-perspective specification, design and implementation of components using aspects. *International Journal of Software Engineering and Knowledge Engineering*, 10(6).
- [Harrison et al., 2002] Harrison, W. H., Ossher, H. L., and Tarr, P. L. (2002). Asymmetrically vs. symmetrically organized paradigms for software composition. Technical report, IBM Research Division, published under RC22685.
- [Herrmann, 2002a] Herrmann, S. (2002a). Composable designs with ufa. In *Workshop on Aspect-oriented Modelling (held with AOSD 2002), Enschede, The Netherlands, 2002*.
- [Herrmann, 2002b] Herrmann, S. (2002b). Object teams: Improving modularity for crosscutting collaborations. In *Proceedings of Net.ObjectDays*.
- [Herrmann and Mezini, 2001] Herrmann, S. and Mezini, M. (2001). Lua aspectual components: Combining composition styles in the evolvable language lac. In *Reviewed position paper at the Workshop on Advanced Separation of Concerns in Software Engineering, ICSE 2001*.
- [Hürsch and Lopes, 1995] Hürsch, W. L. and Lopes, C. V. (1995). Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA.
- [Katara and Katz, 2003] Katara, M. and Katz, S. (2003). Architectural views of aspects. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 1–10, New York, NY, USA. ACM Press.
- [Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353. Springer-Verlag.
- [Kiczales et al., 1997] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J., and Irwin, J. (1997). Aspect-oriented programming. *Proceedings European Conference on Object-Oriented Programming, Springer-Verslag*, 1241:220–242.

BIBLIOGRAPHY

- [Kowalczykiewicz and Weiss, 2002] Kowalczykiewicz, K. and Weiss, D. (2002). Traceability: Taming uncontrolled change in software development.
- [Lieberherr et al., 1999] Lieberherr, K., Lorenz, D. H., and Mezini, M. (1999). Programming with aspectual components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA 02115.
- [Muller, 2004] Muller, A. (2004). Reusing functional aspects : from composition to parameterization. In *Aspect-Oriented Modeling Workshop (AOM 2004)*, Lisbon, Portugal.
- [Ossher and Tarr, 2000] Ossher, H. and Tarr, P. (2000). Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer.
- [Pawlak et al., 2002] Pawlak, R., Duchien, L., Florin, G., Legond-Aubry, F., Seinturier, L., and Martelli, L. (2002). A UML notation for aspect-oriented software design.
- [Pinto et al., 2003] Pinto, M., Fuentes, L., and Troya, J. M. (2003). Daopadl: 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.
- [Pinto et al., 2005] Pinto, M., Fuentes, L., and Troya, J. M. (2005). A dynamic component and aspect-oriented platform. *Comput. J.*, 48(4):401–420. vooral sectie 3 over CAM is interessant voor het design aspect.
- [Rashid et al., 2006] Rashid, A., Moreira, A., Araujo, J., Clements, P., Baniasad, E., and Tekinerdogan, B. (2006). Early aspects: Aspect-oriented requirements engineering and architecture design. <http://www.early-aspects.net/>. This is an electronic document. Date retrieved: January 5, 2006.
- [Rumbaugh et al., 2005] Rumbaugh, J., Jacobson, I., and Booch, G. (2005). *The Unified Modeling Language Reference Manual, Second Edition*. Addison-Wesley.
- [Stein et al., 2002a] Stein, D., Hanenberg, S., and Unland, R. (2002a). On representing join points in the UML. In Kandé, M., Aldawud, O., Booch, G., and Harrison, B., editors, *Workshop on Aspect-Oriented Modeling with UML*.
- [Stein et al., 2002b] Stein, D., Hanenberg, S., and Unland, R. (2002b). A uml-based aspect-oriented design notation for aspectj. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 106–112, New York, NY, USA. ACM Press.
- [Stein, 2002] Stein, D., H. S. U. R. (2002). Designing aspect-oriented crosscutting in uml. In *1st International Workshop on Aspect-Oriented Modeling with UML, AOSD 2002, Enschede, The Netherlands*.
- [Wagelaar, 2002] Wagelaar, D., B. L. (2002). Using a concept-based approach to aspect-oriented software design. AOSD 2002 workshop on "Aspect Oriented Design", position paper for , Enschede, The Netherlands, April 2002.