Components + Aspects : A General Overview [*]

A. M. Reina [†] J. Torres [‡]

Abstract

In the last few years, new ways of decomposing systems have been proposed. First, component-oriented development has been widely recognized as a paradigm for developing systems using pieces called components. But more recently, a new philosophy known as advanced separation of concerns or aspect-oriented programming has arisen. This paradigm has as one of its main aims the improvement of systems' decomposition. Although at first sight it seems that both approaches clash, they are not incompatible. Therefore, the main goal of this paper is to analise the different proposals to bridge the gap between components and aspects. After surveying them, it can be noticed that most of them still are at the implementation level, and there is a lot of ongoing work on earlier phases. Finally, it should be stressed that there is also a great need for metrics in order to measure and compare results in an objective way.

Keywords: *Aspect oriented programming, advanced separation of concerns, component technology.*

Resumen

En los últimos años se han propuesto nuevas formas de descomponer sistemas. En primer lugar, el desarrollo orientado a componentes se ha reconocido ampliamente como un paradigma para construir sistemas utilizando piezas llamadas componentes. Pero, más recientemente, ha surgido con fuerza una nueva filosofía conocida como separación avanzada de conceptos o programación orientada a aspectos. Este paradigma tiene como uno de sus principales objetivos el mejorar la descomposición de sistemas. Aunque a primera vista parezca que ambos enfoques entran en conflicto, éstos no son incompatibles. Por lo tanto, el principal objetivo de este trabajo es analizar las distintas propuestas que hay para acercar el mundo de los componentes al mundo de los aspectos. Tras un análisis de las mismas, se tiene que la mayoría de ellas aún se mueven en la fase de implementación, y que queda mucho por hacer en las fases previas. Además, otro punto a destacar es la necesidad de métricas para medir y comparar resultados de forma objetiva.

Palabras clave: *Programación orientada a aspectos, separación avanzada de conceptos, tecnología de componentes.*

[*]This work has been partially supported by the Spanish Ministry of Science and Tecnhnology and FEDER funds: TIC 2003-369

[†]Languages and Computer System Department. University of Seville, Avda. Reina Mercedes, s/n. Seville, `reinaqu@lsi.us.es`

[‡]Languages and Computer System Department. University of Seville, Avda. Reina Mercedes, s/n. Seville, `jtorres@lsi.us.es`

1 Introduction

In the last few decades, worries about software evolution and reutilization have been growing more and more. Therefore, many proposals have arisen to improve the reuse of software assets. One of the pioneers is component models, which face up to this problem using components as the proposed units of reuse. Thus, according to [37], software components can be seen as executable units of independent production, acquisition, and deployment that can be composed into a functioning system. But recently, a new trend to decompose systems has arisen, known as advanced separation of concerns or aspect-oriented programming [20]. This new proposal has its foundations on trying to solve some anomalies detected in object-oriented languages, such as inheritance anomalies [25], and, also, on facing up to two problems: scattering and tangled code. Furthermore, the principle known as "divide-and-conquer" can be considered as the basis of this new paradigm . So that, it can be stated it is easier to solve a problem if we specify its different concerns or areas of interest. After that, we compose the partial solutions to solve the whole problem.

We can think of component and aspect technologies as two different ways of decomposing a system. Although at first sight it seems that both technologies clash, they are not competing technologies. Thus, currently there is some ongoing research to bridge the gap between both technologies. This paper has two main goals: on the one hand,to be a reference for those researchers who are becoming familiar with aspect-oriented programming and at the same time have a background on component technology, and on the other hand, to identify open issues as a result of surveying the current proposals for integrating crosscutting concerns and component technologies.

After surveying different proposals, we have realised that most of them are at implementation level. Also, at this level, there are two clear trends: firstly, there are approaches which give current component models tools based on aspect-orientation for separating concerns; and, secondly, there are other proposals which state that we should change our mind and think of aspectualize components.

The main contributions of this paper are to pick all the different proposals which mix components and aspects, because due to the novelty of aspect orientation, there are a lack of papers of this kind. As in detecting open issues, for example, we would like to highlight that there is an important need for metrics. The rest is organized as follows: the next section will give a general overview of component technology. After that, an introduction to aspect-oriented programming is given in section 3. Then in section 4 the proposals for mixing aspects and components are introduced. They have been classified in two main groups: proposals focused on implementation (subsection 4.1) and proposals for earlier phases (subsection 4.2). Finally, in section 5, we will enumerate some open issues and conclude the paper.

2 Component Technology

To understand how component technology helps to decompose a system, we have to know which are the main pieces used to build the whole system, components. The term component is very vague in software engineering, and it can have different meanings, thus, in [37], three different, but complementary, definitions of component software are given, each one adopting a different level of abstraction:

- "Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system".

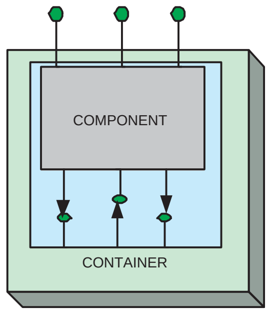


Figure 1: Scheme of a component and its container

- "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties".(Definition given at the 1996 European Conference on Object-Oriented Programming (ECOOP) as one of the results of the Workshop on Component-Oriented Programming).
- "A component is a set of normally simultaneously deployed atomic components. An atomic component is a module and a set of resources".

The definition of a concept, it also can be given using its characteristic properties, thus Szyperski states that a software component:

- is a unit of independent deployment;
- is a unit of third-party composition;
- has no (externally) observable state.

According to the taxonomy proposed in [38] technical components are technical building blocks to assemble applications. These kind of components evaluate their own features by means of a tool known as *container*, which provides a runtime environment for the component. Figure 1 shows the scheme of a component and its container. The container can be thought as a wrapper that will be in charge of dealing with some technical concerns.

Concerns that can be handled by containers are known as *infrastructural services*. They can be concerns such as synchronization, persistence, transactions, security or load balance. The component will have to provide a *technical interface*, in such a way, that all components should have a uniform interface to access to the infrastructure services that the container is providing.

Currently, there are three main models in component software: the one proposed by the Object Management Group, the one proposed by Microsoft, and the one proposed by Sun. Each one, has a component model of the server-side based on container. They are:

Corba Component Model (CCM) [15] CORBA 3 is the last release of CORBA standars, and it proposes the new CORBA Component Model(CCM). CCM is an extension to EJB. CORBA 3 is a container-based specification, that is, every component instance is inside a container.

Component Object Model (COM+) [12] COM+ is an extension of COM, the foundation of Microsoft's component platform. COM+ 2.0 has been used for the .NET Framework. COM+ separates declarative attributes about infrastructure services from the code of components.

Enterprise JavaBeans (EJB) [26] EJB is a Java's component model based on container. The container will implement the runtime environment for the enterprise bean, which includes security, concurrency, life cycle management, transaction and other services.

3 Advanced Separation of Concerns

The separation of concerns is one of the main foundations of software engineering. This foundation is based on the principle known as "divide-and-conquer".Thus, lately, a new paradigm to decompose systems has been brought up. This new proposal, known as advanced separation of concerns (ASC) or aspect-oriented software development (AOSD), states that it is easier to develop programs if we specify their different concerns or areas of interests, and after that, at a later stage, we compose the partial solutions to solve the whole system. Thus, getting a "clean" separation of concerns, will improve comprehensibility and will reduce complexity, because each concern is managed in the right place.

This new paradigm arose trying to solve two problems that are observable while programming, they are:

Scattering This term is referred to the fact that there are certain concerns that aren't well localized, and they are spread throughout the program code. A clear example is the concept known as trace. If a call method is wanted to be traced then the method will have to be dirty adding the code needed to print out a message.

Tangled code This term is referred to those concerns that are not well-localized, and, therefore, they are intermingled producing a tangled code.

Another observable problem, which is addressing the separation of concerns is known as the tyranny of the dominant decomposition. This problem arose because traditional programming languages allow the separation and encapsulation of only one kind of concern at a time. For example, in object-oriented languages the tyrant decompositions are classes while in functional languages, they are functions.

The first proposals for separating things were at implementation level, and they can be considered as the roots of separation of concerns. These proposals are: aspect-oriented programming, subject-oriented programming, composition filters and adaptive programming, and are going to be detailed in the next subsection.

The first papers published in this area of research used the term *aspect* to name the concerns that spread for the whole code program. Thus, one intuitive definition of the term aspect, and also one of the firsts, was given in [20]. The definition of aspect was made comparing the term aspect to the term component: *"a property is a component, if it can be cleanly encapsulated in a generalized procedure, while a property is an aspect if it can not be cleanly encapsulated in a generalized procedure"*. And, also, in [11] a distintion between the terms concern and aspect is given:

- A concern is a domain used as a decomposition criteria for a system or another domain with that concern. Concerns can be used during analysis, design, implementation and refactoring.

- An aspect is a partial specification of a concept or a set of concepts with respect to a concern.

Due to the fact that the term aspect was adopted by aspect-oriented programming [20], and there are other proposals to separate concerns, a more general term has been coined for naming this kind of concerns: *crosscutting concerns*. The term aspect is too related to one proposal and many researchers prefer the use of the term crosscutting concern, to be independent of the approach.

The following subsection is going to introduce the state of the art of separation of concerns at the implementation level, because the first approaches for separating things were at implementation time, proposing new languages and languages extensions. After that, researchers have realized that they need to preview these separated artifacts before implementation, that is, to apply engineering processes to develop software. Thus, nowadays there are proposals for applying the separation of concerns at design, and even at analysis and requirements engineering. But, as we have mentioned before, we are going to focus only at implementation level, because the proposals at this level provide the foundations for the other phases and have create a common vocabulary, and also, because the are models for many proposals which mix aspects and components.

3.1 Concerns at implementation

There are different proposals to obtain a clean separation of concerns during program development. They are organised by the number of papers that they produce in specialized conferences and workshops. None of these proposals give a complete separation of concerns, because concerns have to do with components. If these things were totally apart, it is likely that they will not belong to the same system.

The main proposals to separate things are introduced above. One of the main differences among them is the composition mechanism used to compose the whole system.

Aspect-oriented programming [20, 19] Aspect-oriented programming (AOP) is a new programming paradigm which provides a way to decompose programs into *functional components* and *aspects*. Then aspects and functional components are composed to implement the whole system. The process of composing components and aspects is known as *weaving*. AOP is the most popular proposal, and it has introduced some new vocabulary, such as aspect, weaver or join points. The *weaver* is the tool used to compose classes and aspects, and *join points* are those points in the program code where the behavior can be augmented with the aspect behavior, either before or after the join point. This approach has been implemented by means of Aspect/J [31] an extension to Java.

Subject-oriented programming [18, 29] The idea behind subject-oriented programming is to handle different perspectives of the objects that are going to be modelled. This proposal is also known as multidimensional separation of concerns (MDSOC)[29]. The main difference between this proposal and AOP is that MDSOC works with different models and then integrates them. This fact implies that all concerns are equally important. On the other hand, AOP starts modelling the basic functionality and then augment it with aspects. Therefore, basic functionality is more relevant than aspects. The different perspectives are called hyperspaces, and they are composed using subject composition rules.The implementation of this proposal can be achieved using an extension to Java called Hyper/J [28].

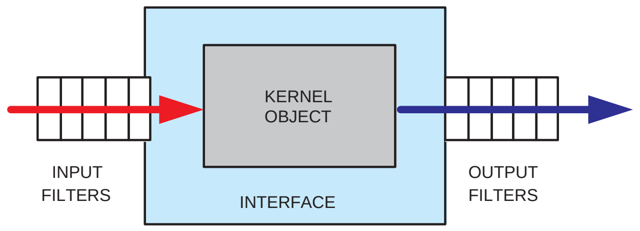


Figure 2: Scheme of an object in the composition filters model

Composition filters [1, 5] Composition filters (CF) are the result of addressing the difficulties of expressing coordination in traditional object-oriented languages. The proposed solution is to augment the object-oriented model with *message filters*. These filters wrap the regular objects. Thus, an object from this model is composed of an *interface layer* (the wrapper) and the *kernel object*. The interface has input filters and output filters, which can modify a message received or sent by the object. Figure 2 depicts a scheme of an object seen by the CF model. The composition mechanisms used in this model are message filters. There are some predefined filters, and new filters can be added, thus, synchronization, error handling, and so on can be captured.

Demeter/Adaptive programming [24, 23] Adaptive programming(AP) was thought of keeping in mind the idea of separating behavior and object structure in object-oriented programs. The composition mechanism of this proposal is known as *traversal strategy*. The class structure is modelled with a graph, and a path to navigate through the class structure has to be given. The traversal strategy is a partial specification of the path mentioned before, because only the initial node of the path and the final node are given, there are no specification about how to reach the end from the initial node. The main advantage of this partial specification is that a modification of the class structure doesn't influence the result of computations.

4 Aspects and Components

One of the pioneering works on applying the separation of concerns ideas to the component world has been [7], where a distinction between contracts according to different negotiable properties is proposed. Some of these properties, such as design by contract or synchronization, have become aspects widely studied by AOP community. Moreover, a separation between control and data constraints is suggested, and as a result, the definition of contracts is completely separated from components which provide the contract-aware services.

Nowadays, one of the main international forums to introduce advances and discuss about open issues and the latest contributions to the combination of aspects and components technologies is the Workshop on Aspects, Components and Patterns for Infrastructure Software (WACPIS), which has been held in conjunction with the International Conference on Aspect-Oriented Software Development (AOSD) since this conference started in 2002.

Many of the papers published in the proceedings of this workshop describe systems that combine components and aspects in various ways. Each system makes different design choices about the nature of the combination; answering questions about the designs can help us to understand the relationships between existing systems and the concerns that they are intended to address.

Implementation	Proposal	Separation Proposal	Component model	Foundations
	JBoss + AOP	AOP	EJB	Mixing current component models and aspects
	WEAVE.NET	AOP	.NET	
	ASPECT C#	AOP		
	Aspect Framework	CF		
	AOP#	AOP		
	CAESAR			New proposals
Design	AOCE			
	UML ACBS			
Requirements	AOCRE			
	ACBSE			

Table 1: Summary of the proposals

One initial common point between both technologies, components and aspects, is what in section 2 has been called *infrastructure services*, because they are one of the most known examples for crosscutting concerns in the advanced separation of concerns (ASOC) world. In [32] an analysis of the pros and cons of modularising infrastructure services with both approaches (components and aspects) is made, and also, a prediction about the future of components is given: "*we envision the next generation of components models to be a set of reusable aspects that can be easily attached to base objects viz. components.*"

Current research on combining aspects and components has two clear directions. On the one hand, there are approaches which try to combine current component models with the ideas proposed in aspect-oriented programming. On the other hand, there are other approaches which propose new models combining components and aspect ideas. These new proposals can be seen as a way of aspectualizing components or componentizing aspects.

Another important focus of attention is software engineering dealing with aspects and components. As it has happened with research on separation of concerns, the previous proposals have been focused on the implementation level. But the need of giving an engineering perspective has strongly arisen, and there are only a few proposals which deal with aspects and components at requirements and design. Table 1 shows a table which summarises the different proposals that are going to be surveyed in this section.

In this table, a classification of the proposals depending on the phase of the development cycle (requirements, design or implementation) is made. But, as it can be seen, there are only a few proposals centered on earlier phases.

As there are a great number of proposals at the implementation phase, we have also made a division of the lines that are apart from the current component models and propose new ways of dealing with aspects and components, and those based on current component models. It must be stressed that there are no proposals based on the CORBA Component Model, and also, most of them have their roots on aspect-oriented programming (AOP) as the way of separating things. Finally, there is one proposal based on composition filters(CF).

The following subsections are going to give more details about the most interesting proposals summarised in the table shown in Figure 1. Thus, section 4.1 introduces the different proposals at implementation level and section 4.2 is devoted to the proposals centered on phases previous to implementation.

4.1 Concerns and components at implementation

Most of the proposals are at the implementation level, but you can find proposals that are providing current component models with aspect-oriented constructors and tools (subsections 4.1.1 and 4.1.2), and other ones which state that current approaches for dealing with aspects and components don't fit well for combining these two paradigms, and they propose new languages and ideas (subsection 4.1.3).

4.1.1 AOP and EJB

Enterprise JavaBeans is a server-side component model which adds abstractions, such as the deployment descriptors, to obtain a clean separation of some concerns such as synchronization, persistence, transaction, or security and the "business logic". In [32] it is stated that AOP is "*a promising approach to eliminate important shortcomings of the container-based component approach*".

In [21], a practical experience developing an application with EJB is introduced, and the conclusions obtained agree with some of the ones obtained in [32]. They are: The container of EJB gives support to some infrastructural services, such as security, performance and so on, and one of the advantages of this approach is the standardization of these services for every container compliant to the EJB specification, but, on the contrary, if we need to add new services to a container, or to configure the infrastructural services, then there is no flexible way or no mechanism to do it. Thus, AOP can be used to add infrastructural services to base objects in a straightforward and transparent way.

Another important advantage of EJB is that the mechanism used to associate the infrastructural services with the EJB is declarative, and at deployment time. On the other hand, one EJB developer has to follow certain rules and there are a set of restrictions that he has to obey, for example, an EJB must not attempt to manage threads. These rules are not enforced and cannot always be checked by the compiler.

Finally, only the classes which live in the context of an EJB can take advantage of the container's services.

There are many vendors of EJB servers, also known as *application servers*, such as WebLogic, WebSphere, JRun, Oracle 9i Application Server or iPlanet. But JBoss has been the first one in taking the initiative of joining to AOP community.

AOP and JBoss JBoss [27] is and open source EJB server and JBoss 4.0 comes with an AOP framework. This framework is inspired by ideas proposed by Kiczales, and it adapts the AspectJ constructors (advices, introductions and pointcuts), adding a new one, metadata.

JBoss architecture has different layers. Figure 3 shows a scheme of the different layers defined in JBoss architercture. The three inner layers (microkernel, service and aspects) are JBoss layers. The application layer is an application written in Java.

The aspect layer will be in charge of mixing the application layer and the inner layers, and the programmer will be allowed to plug or unplug aspects onto objects. Thus, remote, secured or persisted objects can be created.

4.1.2 AOP and .NET

There are a few projects related to the .NET framework. Table 2 shows a brief description of all of them. But for space and clarity reasons, we are going to give details of those proposals which are more related to components frameworks (the Aspect Framework and AOP♯).

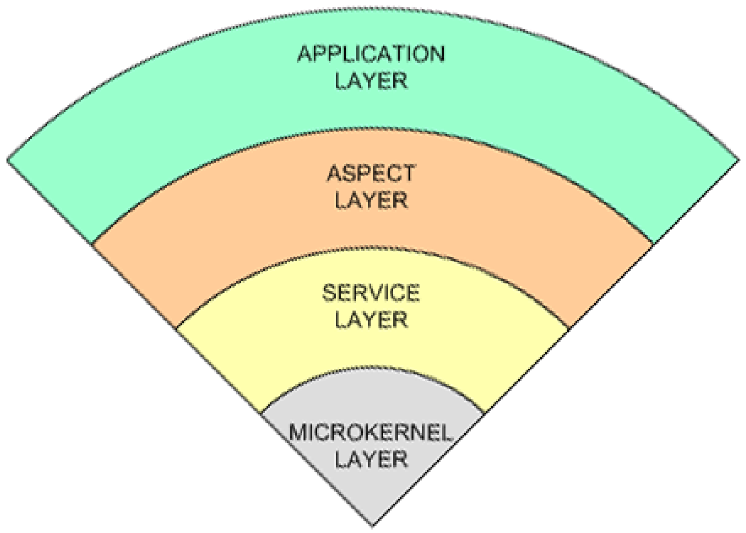


Figure 3: Layers defined in JBoss architecture

Proposal	Reference	Description
WEAVE.NET	[22]	It tries to implement a weaver in such a way that it is language-independent, and based on Aspect/J model.
Aspect C♯	[33]	It tries to extend the compiler avalaible under Microsoft's Shared Source Common Language Infrastructure (SSCLI).
Aspect Framework	[36]	It is an aspect framework for COM components and for the .NET framework.
AOP♯	[35]	It is an AOP implementation on the .NET.

Table 2: Proposals based on the .NET component model

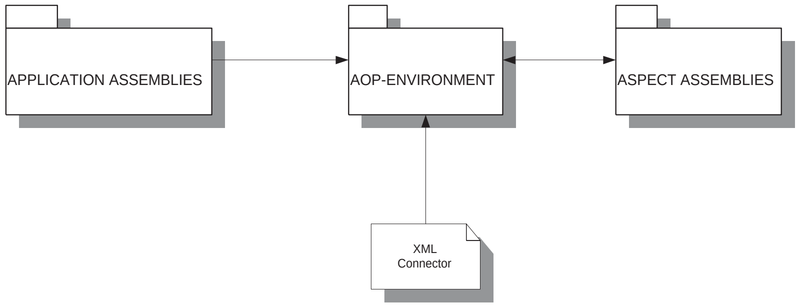


Figure 4: AOP♯ overview

Aspect Framework Although COM+ provides many infrastructure services, it lacks of the ability to define customized aspects, the same as EJB. In [36] an aspect framework to develop systems with personalized aspects is given.

The idea behind this proposal was borrowed from composition filters (CF). The framework implements user-defined aspects as COM components, and associates aspects and components with metadata descriptions.

The key mechanisms used to implement the framework are interception and delegation. Thus, there are only two steps that the framework has to give: activation and method invocation.

When a component is activated, the framework builds a stack with all the aspect object implementation and returns a reference to the interceptor. The interceptor delegates the calls to all the registered aspects for pre-processing. After that, it delivers the actual call to the object. Finally, it delivers the call to the aspects for post-processing. Thus, one aspect in the framework is a COM object which implements the interface `IAspect`.

The association between aspects and components is done by means of an XML file.

The same solution has been applied to the .NET framework, but with the differences related to the context. For example, the .NET proposal associates aspects and components via attributes. Aspects here are components which implement the interface `IMessageSink`.

AOP♯ In [35] AOP♯ (an AOP system for Microsoft's .NET platform) is proposed.This proposal has been thought having three requirements in mind:

1. No language extension.

2. Complete separation of aspects and basic functionality.

3. Provide an easy mechanism to compose aspects and basic functionality.

A program written for the AOP♯ platform with .NET has three parts: the *application assemblies*, where business rules are implemented; the *aspect assemblies*, where aspects are encapsulated, and the *AOPEnvironment*, which is provided by the AOP system, and it will be in charge of mixing aspects and business rules at runtime. This composition is specified using an XML file called *connector*. Figure 4 obtained from [35] gives general view of the proposal.

The most interesting characteristic of this proposal is the fact that aspects can be enabled or disabled at runtime. Thus, the behavior of objects can be changed depending on the aspects that are being applied.

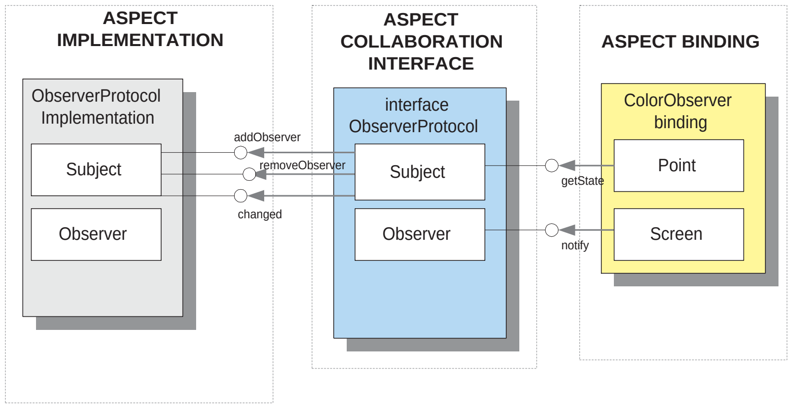


Figure 5: Caesar model

4.1.3 Aspectualizing components

After doing an analysis of current AOP proposals and component models [32], some drawbacks were found:

- Languages of separation of concerns don't modularize infrastructural services well because they don't provide a black-box way of reusing and they are based on code transformation.
- Reusing aspects. The reuse of aspects requires the ability of the programmer.
- Aspects at deployment time. AspectJ requires that aspects are applied to base objects at design time, and the application of aspects couldn't be postponed to deployment time.

Thus, trying to address these drawbacks, some requirements have been asked for solving the previous points, they are:

- *Aspectual polymorphism.*
  This is a late binding, that is, the reusable component will have operations late bound depending on the runtime context of the component. This requirement is key to bridge the gap between adaptability and black-box reuse.
- *Separate infrastructure models.*
  This means that component models should be aspectualized by providing appropriate linguistic means to capture individual infrastructure services in separate modules.

In order to fulfill these requirements, a new model for aspect-oriented programming called Caesar [30] has been proposed. One of the most interesting ideas proposed in this approach is the separation of aspect implementations and aspect bindings, which allows better reutilization of aspects.

The model has reached this separation using Aspect Collaboration Interfaces (ACI), whose purpose is decoupling aspect implementations and aspect bindings. Figure 5 depicts the three main elements of the Caesar model. Looking at this figure we can realize how the implementation of an aspect is decoupled from bindings.

Figure 5 represents a scheme of Observer pattern's implementation using CAESAR. This scheme corresponds to the example implemented in [30]. The *Aspect Collaboration Interface* consists of several mutually recursive nested ACI's (one for each abstraction in the modular structure of the aspect). Thus, in the figure we can see two ACI's nested, `Subject` and `Observer` inside the ACI called interface `ObserverProtocol`. The ACI's will specify the provided facet of the aspect, that is, what the aspect will provide to the context in which it will be applied. In our example, the provided methods are `addObserver`, `removeObserver` and `changed`.

On the other hand, the ACI will also expect something from the context to which it will be applied. The expected methods from our example are `getState` and `notify`.

The aspect implementation has to implement all the provided methods specified in the ACI, while the aspect binding will implement the expected methods.

4.2 Software engineering with aspects and components

The earlier subsections explain proposals too focused on implementation problems, but, for developing systems we need some engineering process. The following sections are going to introduce some proposals for applying aspects and components to software engineering.

4.2.1 AOCE

In AOCE (Aspect Oriented Component Engineering)[17], and previously AOCRE (Aspect Oriented Component Requirement Engineering) [16], Grundy proposes a process for developing aspect-oriented component applications from the first stages of software development. We can see AOCRE as the first stage of AOCE. Grundy defines aspects as "*horizontal slices of a system's functionality and non-functional constraints, and include user interfaces, collaborative work facilities, persistence and distribution management, and security services*". AOCE avoids weaving and uses reflection at runtime.

To understand this proposal, we need to give some definitions:

A component *provides* certain services related to aspects for other components to use, and *requires* one or more services from other components. A categorization of the different services that components provide and require is made.

This pioneering proposal in the area of aspect oriented software engineering introduces a different perspective of aspects, in the sense that while most of the proposals think of aspects as they are depicted in Figure 6 (a); Grundy has the conception shown in Figure 6(b).

AOCRE starts with a general requirement analysis, and continues with an iterative refinement of requirements. Its stages are the following:

1. Identify component candidates.

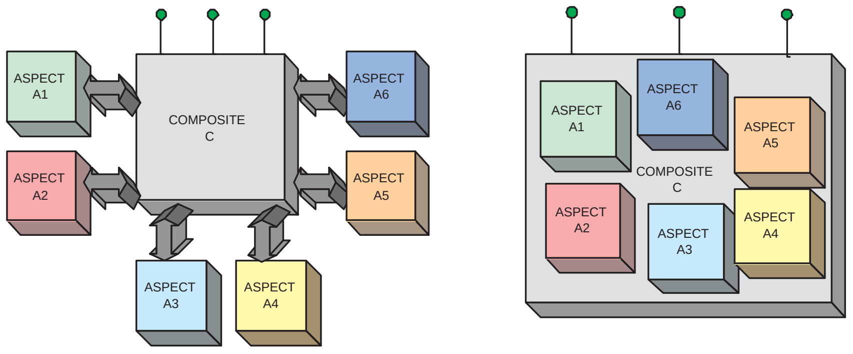
   Grundy proposes to find component candidates using object-oriented analysis diagrams or using components from inverse engineering .

2. For each component, identify aspects.

   At this stage, aspects for which the component provides services and aspects that are provided by other components

3. Refine aspects, indicating aspects provided and aspects required for each component.

   Grundy introduces the term *aspect details* to describe more precisely certain component's characteristics that has to do with the aspect.



(a) General view (b) Grundy's view

Figure 6: Aspect perspectives

4. Analyze aggregated aspects.

   Aggregated aspects are a group of aspects related to a group of components.

5. Verify requirements found.

   Grundy proposes a textual specification of components,aspects and required and provided aspects.

Figure 7 shows a scheme of the basic AOCRE process obtained from [16].

4.2.2 ACBSE

Aspect Component Based Software Engineering (ACBSE) [8] is an extension to Component-based Software Engineering (CBSE). Thus, it is based on the phases proposed for component-based life cycle:

1. Interface specification.
2. Component specification
3. Component implementation
4. Package
5. Assembly and deployment

During the two first stages, the components' provided and required interfaces must be described. This approach proposes, at this point, do a classification of the component dependencies. Thus, they can be catalogued in two different types: intrinsic and non-intrinsics.

This classification has one goal, to postpone crosscutting concerns to package phase, because at the implementation phase, only intrinsic dependencies are going to be used. Intrinsic dependencies are those which are crucial for the component, without them, the component couldn't live.

This proposal starts with the design phase, but it doesn't worry about requirement nor analysis phases. In [9], an approach for modelling components and aspects with UML is proposed. Thus an extension to UML is made by means of stereotypes, one for each kind of dependency.

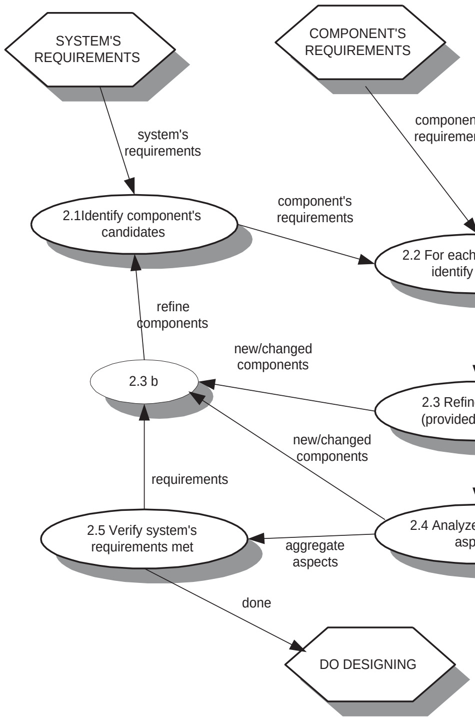


Figure 7: Basic AOCRE process

5 Open Issues and Concluding Remarks

As the advanced separation of concerns is a relatively young area of research, there still are many things to do. From our experience comparing AOP proposals with others [34], we think that there are some open issues that will have significance in the near future. On the one hand, we need metrics to check the goodness of an aop solution. As we think that this is one of the points where there is much ongoing work, we will give more details about it in subsection 5.1.

If we look at the proposals surveyed in the previous section, we will realize that most of them are at the implementation level, and there are only a few ones at earlier phases. Now there is a growing community of researchers devoted to what is known as early aspects, that is, the detection of crosscutting concerns in the first stages of development of software projects. In this line, an important forum of discussion is the Workshop on Early Aspects [4, 2]held in conjunction with the Conference on Aspect-Oriented Software Development. But most of the approaches are centered on a traditional, non component-oriented development process.

Finally, as aspects have been thought to improve reuse, there are some proposals [14] to build aspect's repositories, but we need some criteria to look for aspects in the repository. Are the criteria used in looking for components also useful in looking for aspects? In this sense, we think that the same questions proposed to search out components are totally applicable for aspects. They are:

- How can someone find one concern needed in his application?
- Which are the relevant characteristics of an aspects in order to be found by someone?
- How can I configure an aspect to be plugged into an application?

5.1 Metrics

Separation of concerns is a philosophy which is growing and evolving at a very high rate. It has been considered one of the ten most important emerging technologies in [13]. But, now, that aspect-oriented development is real, we need some criteria to measure the goodness of the developments.

There is a logical reason for which metrics haven't been proposed yet: many metrics arise from experiences using and applying some specific technology, and, there have been very little experience using aspect orientation. But now, when aspect-oriented tools are being used for the development of applications, we need metrics to compare two different aspect-oriented solutions which address the same problem. But, we also need to know the advantages of an aspect-oriented solution if we compare it to another non aspect-oriented solution in a rigorous way, that is, we need to quantify these improvements, if there are some.

Therefore, as it is stated in [39],we need metrics for measuring efficiency, understandability, and reusability of an aspect-oriented design. In agreement with this, in [6] a classification of metrics is given according to the objectives that we have when we are going to use specific metrics:

1. Metrics for comparing traditional techniques to aspect-oriented techniques.
2. Metrics for comparing the use of static aspects to the use of dynamic aspects.
3. Metrics for measuring the quality of aspects.

In the third category we can include the suite of metrics proposed by Zhao [40], which are designed to quantify the information flows in aspect-oriented programming. These metrics are based on the dependence model of aspect-oriented software, and can be used to measure the complexity of aspect-oriented software. Although Zhao states that these metrics are language-independent, the implementation of a tool to analyze the dependencies can be different from a language to another.

To conclude, we can state that all the proposed metrics analyzed in the different works mentioned in this subsection have been thought to be applied during the implementation phase, but there is much more work to do for quantifying things during earlier phases. If we focus on implementation, all the metrics are applied to system developed with AspectJ, but we think that they should be proved with other proposals of separation of concerns.

Finally we need to do more research to compare separation of concerns proposals with traditional applications.

5.2 Final remarks

This paper has given a general overview of the state of the art of the aspect and component technologies, focusing mainly on those proposals which combine both approaches. Separation of concerns technology is currently at a point very similar to the point in which object-oriented technology was more than twenty years ago. Therefore, there are many open issues and a lot of work to do.

Looking at all the proposals enumerated in section 4, we realize that most of them still are centered at the implementation level. But, an engineering process should be applied, that is, concerns should be previewed from the very beginning.

Related to the implementation phase, we have noticed two clear trends: on the one hand, those which pick up a component model and a proposal of separation, and mix both; but, on the other hand, there are some voices telling that current aspect languages aren't powerful enough, and that new proposals to aspectualize components are needed. Finally, metrics are needed to check and compare results objectively.

References

[1] M. Akşit, J. Bosch, W. V. D. Sterren, and L. Bergmans. Real-time specification inheritance anomalies and real-time filters. In M. Tokoro and R. Pareschi, editors, *Proc. 8th European Conf. Object-Oriented Programming*, pages 386–407. Springer Verlag LNCS 821, July 1994.

[2] *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (AOSD-2002)*, March 2002.

[3] *First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-2002)*, March 2002.

[4] J. Araújo, A. Rashid, B. Tekinerdogan, A. Moreira, and P. Clements, editors. *Early Aspects 2003: Aspect-Oriented Requirements Engineering and Architecture Design*, March 2003.

[5] L. Bergmans and M. Akşit. Composing crosscutting concerns using composition filters. *Comm. ACM*, 44(10):51–57, October 2001.

[6] M. F. Bertoa and A. Vallecillo. Reflexiones sobre la definición de métricas para software orientado a aspectos. In J. Hernández, L. Lozano, and A. Moreira, editors, *Proceedings of Taller de Desarrollo de Software Orientado a Aspectos*, Nov 2003.

[7] A. Beugnard, J. M. Jezquel, N. Plouzeau, and D. Watkins. Making components contracts aware. *IEEE Computer*, 32(7):38–45, 1999.

[8] P. J. Clemente and J. Hernández. Aspect component based software engineering. In Coady et al. [10].

[9] P. J. Clemente, F. Sánchez, and M.A. Pérez. Modeling with uml component-based and aspect oriented programming systems. In *Proceedings of the Seventh Workshop on Component-oriented Programming. Held in conjunction with ECOOP 2002*, Jun 2002.

[10] Y. Coady, E. Eide, and D. H. Lorenz, editors. *The Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, March 2003.

[11] K. Czarnecki, U. W. Eisenecker, and P. Steyaert. Beyond objects: Generative programming. In *Workshop on Aspect Oriented Programming (ECOOP 1997)*, June 1997.

[12] G. Eddon and H. Eddon. *Inside COM+*. Microsoft Press, 1999.

[13] Editors. The technology review ten. *MIT Technology Review*, pages 97–113, 2001. January/February.

[14] L. Fuentes, D. Jiménez, and M. Pinto. Hacia un entorno de desarrollo integrado basado en componentes y aspectos. In *Actas del Taller de Trabajo en Desarrollo de Software Orientado a Aspectos*, nov 2003.

[15] Object Management Group. Corba components final submission., 1999.

[16] J. Grundy. Aspect-oriented requirements engineering for component-based software systems. In *4th IEEE International Symposium on Requirements Engineering*, pages 84–91. IEEE Computer Society, 1999.

[17] J. Grundy. Multi-perspective specification, design and implementation of software components using aspects. *International Journal of Software Engineering and Knowledge Engineering*, 19(6), 2000.

[18] W. Harrison and H. Ossher. Subject-oriented programming—a critique of pure objects. In *Proc. 1993 Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428, September 1993.

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Comm. ACM*, 44(10):59–65, October 2001.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[21] H. Kim and S. Clarke. The relevance of AOP to an applications programmer in an EJB environment. In AOSD-PAT02 [3].

[22] D. Lafferty and V. Cahill. Language independent aspect-oriented programming. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, oct 2003.

[23] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. *Comm. ACM*, 44(10):39–41, October 2001.

[24] K. J. Lieberherr. *Adaptive Object-Oriented Software: the Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, 1996.

[25] S. Matsuoka and A. Yonezawa. Inheritance anomaly in object-oriented concurrent programming languages. *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150, 1993.

[26] Sun Microsystems. Enterprise javabeans specification, version 2.0, 2001.

[27] JBoss Org. Jboss web page: http://www.jboss.org.

[28] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proc. Symposium on Software Architectures and Component Technology: The State of the Art in Software Development.* Kluwer, 2000.

[29] H. Ossher and P. Tarr. The shape of things to come: Using multi-dimensional separation of concerns with Hyper/J to (re)shape evolving software. *Comm. ACM*, 44(10):43–50, October 2001.

[30] K. Ostermann and M. Mezini. Conquering aspects with Caesar. In M. Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 90–99. ACM Press, March 2003.

[31] Xerox PARC. Aspectj home page. web, 2002.

[32] R. Pichler, K. Ostermann, and M. Mezini. On aspectualizing component models. *Software Practice and Experience*, 33(10):957–974, 2003.

[33] M. Devi Prasad and B. D. Chaudhary. AOP support for C#. In Coady et al. [10].

[34] A. M. Reina, J. Torres, M. Toro, and J.A. lvarez. Concerns vs. components for web development. In *Proceedings of the IADIS WWW/Internet 2003 Conference*, pages 873–876. IADIS Press, 2003.

[35] M. Schüpany, C. Schwanninger, and E. Wuchner. Aspect-oriented programming for .NET. In AOSD-PAT02 [3].

[36] D. Shukla, S. Fell, and C. Sells. Aspect-oriented programming enables better code encapsulation and reuse. *MSDN Magazine*, 17(3), 2002.

[37] C. Szyperski, D. Gruntz, and S. Murer. *Component Software. Beyond Object-Oriented Programming.* Addison-Wesley, 2002.

[38] M. Voelter. A taxonomy of components. *Journal of Object Technology*, 2(4):119–125, 2003.

[39] A. A. Zakaria and H. Hosny. Metrics for aspect-oriented software design. In O. Aldawud, M. Kandé, G. Booch, B. Harrison, and D. Stein, editors, *Third International Workshop on Aspect Oriented Modeling*, March 2003.

[40] J. Zhao. Towards a metric suite for aspect-oriented software. Technical Report SE-136-25, Information Processing Society of Japan, 2002.