

# Agility and the AOP paradigm - Flexibility of Software, Services and Processes

Matthias Wiselka

Universität Stuttgart

wiselkms@studi.informatik.uni-stuttgart.de

<http://www.uni-stuttgart.de>

**Abstract.** The 1997 introduced aspect-oriented programming paradigm promises to increase flexibility of software by increasing modularity through the separation of crosscutting concerns. We will show how AOP is aiming to keep its promise by giving an overview of its concepts, explained in a simple example. However, AOP is not only limited to the software level. With the emergence of service-oriented application development (SOAD), the adaptability of services and processes has gained more and more importance. AOP offers the concepts and techniques to improve adaptability needs, as we will show. We will give an overview of why software adaptability is needed and present an approach based on web service technologies applying AOP. Subsequently, we are going to discuss process adaptability as well. A classification of approaches will be given and this paper will present three concrete approaches for the de facto standard BPEL, showing that aspect-oriented programming can not only increase the agility of programming in the small, but also of programming in the large.

**Key words:** Agility, Aspect-Oriented Programming, Service Adaptability, Process Adaptability, BPEL, Web Service

## 1 Introduction

Since the introduction of OOP, computer programs have the goal to be modular and flexible enough to be reused where needed. However, OOP couldn't satisfy this promise completely. There are some requirements to software that can't be decomposed to well encapsulated modules with OOP as needed, hereby, coupling software unnecessarily tight. In 1997, the aspect-oriented programming paradigm was introduced to the world by the team of Gregor Kiczales with the goal, not to replace OOP, but to complement it. By allowing the separation of the concerns that made problems with OOP, AOP promises to make software more flexible and increase reuse.

However, AOP isn't just limited to software, but can also be applied to service-oriented environments. The technologies offered today are not flexible enough to meet business needs, where requirements are changing quickly and the corresponding implementations need to be adjusted accordingly. Web services are usually hardwired with applications and can't be quickly adapted to

others without great effort. Business processes implemented as WS-flows are usually modeled statically and can't be changed once they are deployed and being executed. This lack of adaptability can be added through the application of the aspect-oriented programming paradigm, as we are going to show.

Section 2 will give a brief introduction in the paradigm of aspect-oriented programming, explaining the concepts in detail and providing a small and simple example in AspectJ. The following section 3 will show the need for adaptability concerning the handling of web services and what role AOP can play in this context. Finally, section 4 will discuss what adaptability means in a process-based environment with the focus on BPEL. A classification of approaches will be discussed and several concrete approaches using AOP will be presented showing how AOP can increase the adaptability of processes, particularly during runtime.

## 2 Aspect-Oriented Programming

### 2.1 Motivation

In the early 1990s, the object-oriented programming (OOP) changed the world of software development and succeeded the procedural programming because it fit better for real domain problems. Inheritance, abstraction and encapsulation are just a few features of how OOP fundamentally helped software engineering. As great as OOP was at first, however, there were still some programming problems left that couldn't be solved sufficiently, neither with the object-oriented nor the procedural approach. One problem, for instance, is that you can't encapsulate certain requirements completely into separate modules, e.g. logging or persistence. Despite the application of OOP, one of these requirements is usually scattered all over the code making the entire code difficult to maintain and coupling it unnecessarily to a specific environment, therefore, complicating the reuse in a different environment. In 1997, Gregor Kiczales and his team at the Xerox Palo Alto Research Center introduced in their paper [11] the aspect-oriented programming (AOP) paradigm, an approach to solve these problems of OOP. The very same team developed later the most popular AOP language called AspectJ<sup>1</sup> that eventually became a part of the Eclipse Community.

### 2.2 Basics

Understanding how AOP works and why it benefits programming requires first understanding its targets. Any computer program can be defined by a set of so-called concerns. "A concern is some functionality or requirement necessary in a system, which has been implemented in a code structure", defined by [7]. In OOP you would usually express a concern, e.g. for the handling of a bank account, by a class that combines the necessary data and functionality.

---

<sup>1</sup> The AspectJ Project: <http://eclipse.org/aspectj>

Through the separation of concerns in a structured software development, as introduced by Edsger Dijkstra [5], a software program is logically separated into components, thus, reducing the complexity of the designed software. In an ideal world the separate concerns would be independent from each other, and therefore, minimally coupled, increasing the chances of being reused. In the real world, on the other hand, there are some concerns that usually cross other concerns. Data logging is a common example for such a concern that would be used not only in the logging module but also across many other parts of a program. These concerns were named *crosscutting concerns* by Kiczales.

*Example 1.* Simple Java class showing the problematic of a crosscutting concern (logging statement) intertwined with a core concern (*hello world* statement).

```
public class HelloWorld {

    public static void main(String[] args) {
        // logging code
        System.out.println("log: invocation of main method");

        // actual method code
        System.out.println("Hello, world!");
    }
}
```

The following example - shown in fig. 1 & 2 - highlights the difference between a core concern and a crosscutting concern on a real life example, the open source servlet container Tomcat of the Apache foundation<sup>2</sup>. The figures show Tomcat's module structure and module sizes in LOC. The red bars indicate the position and the size of a concern in a module. Fig. 1 shows the part of the code that handles the XML parsing. As you can see, it can be found concentrated within a single module and it is entirely decoupled from the other modules. In fig. 2, on the other hand, you can observe the code for the logging in Tomcat. The logging concern can't be assigned to a single module but is rather scattered in all modules. It is *crosscutting* the other concerns, hence, increasing the tangling of the code and narrowing the reuse and ease of maintenance.

The paradigm of aspect-oriented programming allows increasing modularity of software by allowing the separation of crosscutting concerns. AOP supports the programmer in expressing and encapsulating crosscutting concerns in own modules, so-called *aspects*, separate from the components, and therefore increasing the reuse and flexibility of code within a program by loosening the coupling between modules. In a next step, the so-called *weaving process* links the aspects with the relevant joinpoints, either on source code level or during run-time.

<sup>2</sup> The Apache Software Foundation: Apache Tomcat: <http://tomcat.apache.org>

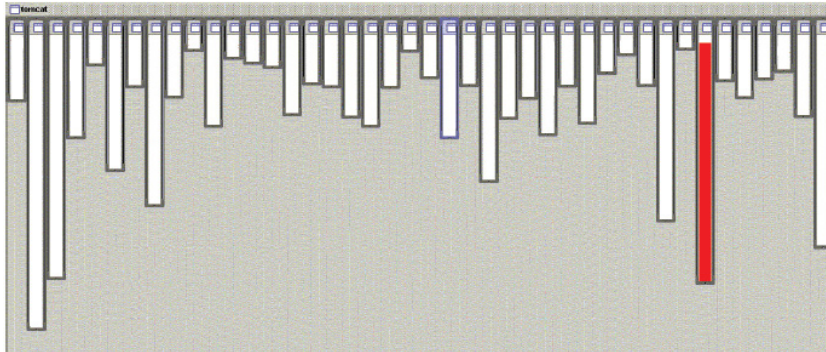


Fig. 1. XML parsing concern in Tomcat as in [2]

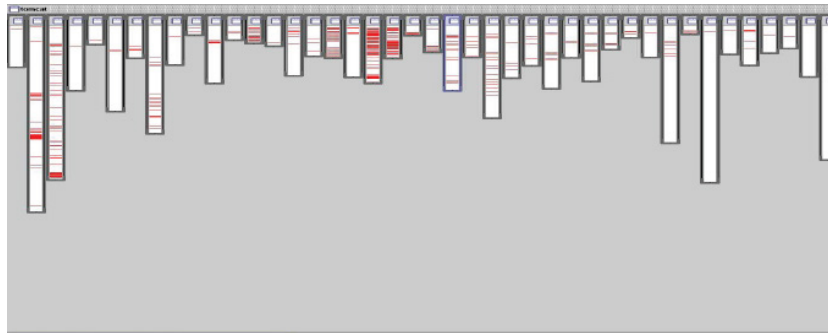


Fig. 2. Logging concern in Tomcat as in [2]

### 2.3 Concepts

This section will show how AOP defines and maintains its task. An example is going to be presented written in AspectJ 5, an AOP extension to the Java programming language.

#### Aspect.

*A property that must be implemented is an **aspect**, if it can not be cleanly encapsulated in a generalized procedure.* (Kiczales [11])

Whereas a class is the fundament of OOP, an aspect fulfills this role for AOP. It represents the implementation of a crosscutting concern and combines the concrete implementation (advice) as well as the information to where the aspect is going to be applied (pointcut).

**Joinpoint.** A joinpoint is a well-defined point in the control flow of a component, which the programmer can choose to apply to a crosscutting concern.

Therefore, the programmer can add additional functionality to a point that wasn't there before.

Joinpoints (as defined in AspectJ) can be:

- An invocation of a method
- An execution of a method
- The handling of an exception
- An access of a variable
- The initialization of a class or object

*Example 2.* Code from example 1 cleaned of crosscutting concerns, which shows where a possible joinpoint could be to apply our logging code to.

```
public class HelloWorld {

    public static void main(String[] args) {
        // possible joinpoint: on execution

        // actual method code
        System.out.println("Hello, world!");
    }
}
```

**Pointcut.** A pointcut is a set of joinpoints. It is part of an aspect and defines the points where the functionality of an aspect needs to be applied to. Through the use of regular expressions in AspectJ, pointcuts also allow the quantification of joinpoints.

**Advice.** An advice contains the actual functionality of an aspect telling what to do when applied at the defined pointcuts. It also gives the possibility to choose the moment when to apply the advice: either before, after or even instead of a joinpoint.

*Example 3.* The complete aspect written in AspectJ defining a pointcut on execution of the main method from the HelloWorld class and the advice to execute a logging statement before the execution of the main method. The logging crosscutting concern from example 1 is completely encapsulated in this aspect code and separated from the core concern code of example 2.

```
public aspect HelloWorldAspect {
    public pointcut mainMethod():
        execution(public static void HelloWorld.main(String[]));

    before(): mainMethod() {
        // logging statement
        System.out.println("log: invocation of main method");
    }
}
```

**Weaving.** Having the source code for core concerns and the aspects for the crosscutting concerns the only task left is to compose these together to create a running program. The *aspect weaver* does this part of AOP in the so-called *weaving process*. There are two classifications for weaving mechanisms: *static* and *dynamic*.

Using static weaving, aspects are weaved into the components before compilation, linking them permanently to the components at run-time. For this kind of weaving, a compiler needs to be extended by a special pre-compiler, which modifies classes by inserting advices at the selected joinpoints yet producing compiler-compliant code. The result of this process is a highly optimized woven code whose execution performance is comparable to that of code written by traditional methodologies [6]. However, a downside of static weaving is the fact that aspects cannot be changed or influenced during run-time [12]. Static weaving is used e.g. by AspectJ. Using the code from the examples 2 & 3 the weaver of AspectJ would generate code that is semantically equivalent to the code from example 1.

Dynamic weaving, on the other hand, allows the de-/activation or interchangeability of aspects during run-time [12]. There are different approaches to accomplish this goal. One that doesn't modify the source code uses so-called *interceptors*, which recognize the reach of a joinpoint during run-time. For this solution we need a modified virtual machine or interpreter depending on the language used. Although, dynamic weaving has many benefits, it usually causes a worse run-time performance.

## 2.4 Summary

The separation of core concerns and crosscutting concerns brings a higher flexibility to software. Modules are less coupled, and less dependent on environment specifics. Yet, this flexibility usually comes with a loss of performance and has to be considered before deciding for AOP. Typical requirements, which can be implemented optimally with AOP, are e.g. logging mechanisms and security features.

As shown earlier, the aspect-oriented programming is not replacing object-oriented programming but extending it to improve it. Since its introduction in 1997, almost every language has received its own implementation of AOP. AspectJ is the best-known implementation for the Java programming language but there is also Aspect.NET for the .NET Framework, AspectC++ for C++ or even AspectL for List.

Although AOP has been around for over 10 years, and its maturity has increased, only few projects are being developed using AOP. The best-known project using AOP is the Spring Framework<sup>3</sup>. The Spring developers have developed the Spring AOP Framework, which is used by Spring internally for transaction management, security, etc. There are still less industrial but more

<sup>3</sup> Spring Framework: <http://www.springframework.org/>

research projects using AOP. One of the reasons is that there are still a lack of good tools supporting the programmer in his AOP needs.

### 3 Service Adaptability

#### 3.1 Motivation

Using the service-oriented architecture (SOA) companies try to gain flexibility and to improve the interoperability between their own systems, as well as systems from business partners, while continuously adjusting their business processes to changing requirements. A very popular solution for implementing a SOA is the use of web services (WSs). Web services offer loose coupling of services and interoperability in a platform, and language independent, manner. Current approaches for WS-integration require e.g. the hard-wiring of WS-references in applications and hereby limit the flexibility and adaptability of applications. Also, WSs fail to be reusable when they need to comply with service contracts offering features like security or authentication. The web service management layer presented in the following section tries to overcome some shortcomings of the current WS-technology making use of the aspect-oriented programming paradigm as a well suited technology that offers concepts for a great increase of adaptability and flexibility for the collaboration with web services.

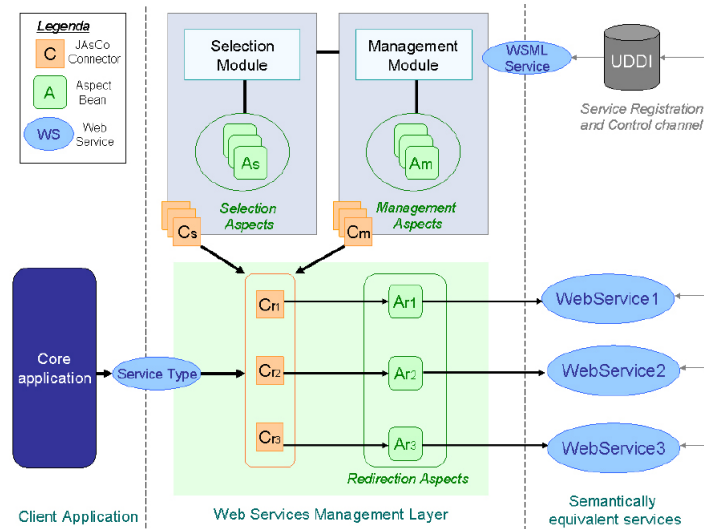
#### 3.2 Web Services Management Layer

The approach of the *WSML* (Web Service Management Layer) introduced in [13] tries to decouple the concrete WSs from the application. In this client-side approach, a WS is not hardwired with the application. Instead of requesting some functionality directly from a WS, the application requests it from the WSML. The WSML can choose, depending on the business requirements, a fitting WS from a collection of semantically equivalent services. The selection can be based on performance, time or costs. If a service were to brake down, the requests could be redirected dynamically to the next best semantically equivalent WS offering the same functionality without restarting or redeploying the application, even if the new WS differs syntactically.

The WSML approach uses the dynamic AOP language JAsCo<sup>4</sup>, which is based on Java. JAsCo allows the dynamic deployment of so-called aspect beans making it possible to weave and unweave new aspects during run-time through *connectors*. The application makes a request to a *service type* in the WSML instead of directly to a WS, see fig. 3. The service type of the WSML hides all WS-specifics from the application, completely decoupling it from the WS-logic. On the invocation of a service type, the WSML chooses one of the known WSs, which fits the best, using a WS selection policy implemented by *selection aspects*. Additionally, *management aspects* can be used to handle management concerns like monitoring and caching. Last but not least, the *redirection aspects*

<sup>4</sup> JAsCo homepage: <http://ssel.vub.ac.be/jasco/>

handle the redirection logic to invoke one or more WSs and prepare the results according to the requested service type.



**Fig. 3.** The detailed architecture of WSML as in [13]

This approach can have a huge increase of flexibility in applications dealing with WSs because the adaptation to new business requirements is continuous and new WSs can be added during execution that were unknown at deployment time. Also, hot-swapping becomes possible and an alternative WS can be used in case a WS is unreachable. Due to the fact, that instead of invoking a concrete WS the application just requests some service functionality in a generic and transparent way, there is no code left handling any WS concerns in the application code itself, thereby increasing the ease of maintenance of the core application.

### 3.3 Summary

As we have seen, the flexibility of services is an important issue that can be fulfilled using AOP. The WSML offers a complete decoupling from WS-logic on the client side, making it possible to switch used WSs dynamically according to needed requirements. If a WS becomes unavailable or if there is a WS that offers similar functionality but for a lesser price, the WS to be used can be changed transparently during run-time without the application knowing it. This approach offers a great gain in flexibility and maintenance. There are other approaches for service adaptability like [1] that address a different need for flexibility of WSs. It offers the possibility of dynamic adaptation of service contracts towards non-functional requirements like security. A policy-compliant web service stub can

be created dynamically through a so-called policy engine for the client as well as the service-side. Overall, service adaptability is an important issue and AOP can greatly increase the interoperability and flexibility of web services.

## 4 Process Adaptability

### 4.1 Motivation

In sections 2 and 3 we have shown that the AOP paradigm can benefit programming in the small. But how can AOP benefit programming in the large? There are several approaches that show AOP can provide flexibility, particularly during run-time, for programming in the large, i.e. in process-based applications. In this context, WSs do not only offer functionality but can be composed also into a process, so-called WS-flow, and offered again as a single WS. This ability makes the web service technology a fitting choice for workflow implementations. Unfortunately, WS-flows inherit, to some extent, the shortcomings of the WS-technology.

The de facto standard for specifying WS-flows is the business process execution language (BPEL) and, hence, is the focus in this paper. A downside to processes described in BPEL is that the adaptability is limited due to the static description [12]. WSs e.g. are described statically through portTypes, and the structure of the control flow is already defined during modeling. BPEL also doesn't offer any support for the modeling of crosscutting concerns. The lack of adaptability in reaction to ever-changing business rules makes processes inflexible and not reusable.

By applying the aspect-oriented programming paradigm to the BPEL environment, it is considered to be a promising approach to add adaptability to processes in BPEL, hence, allowing more flexibility, particularly during execution. For a high acceptance rate, a proposed approach must offer reuse and be non-intrusive with respect to existing technologies and infrastructure, and therefore preserve already executed investments.

### 4.2 Classification of adaptability

Due to the continuous changes of companies and their business processes workflows have to be flexible and able to evolve in their life cycle. Evolution means in this context that the schema is adjusted to new requirements. For an adaptable process, we expect it to be able to change its performing tasks, as well as the execution order of these tasks, but also the changeability of the used WS types is essential. These changes should be able to be performed not only directly at the schema, but at a concrete instance of a process as well. There are several approaches of how to make processes more flexible and letting them react to modified conditions. By limiting them to the process layer, as in [8], the classification of the different approaches can be greatly simplified to four different groups, as presented in figure 4.

		Level	Approach
Life cycle	Build-time	Model	Any changes possible
		Instance	Pre-planned flexibility
	Run-time	Model	Business logic and WS-portType changes for all existing or future instances
		Instance	Business logic and WS-portType changes for only one or more existing instances

Fig. 4. Classification of approaches to process adaptability based on [8]

Commonly, a process can be adjusted to changes during *build-time*, when it is being modeled or created, but also during *run-time*, when it is being executed. In both cases, changes can affect either the *model* (schema) or a concrete *instance*. During build-time, it is clear that a process can be adapted to any kind of evolution of its environment. This can be accomplished by either remodeling the schema according the new requirements or by providing alternative execution paths for future changes. On deployment, these paths can be selected e.g. by the user. Since future changes aren't usually known, especially not to the full extent, this pre-planned flexibility makes processes not adaptable enough.

Far more interesting is the adaptability of processes during execution because usually the evolution takes place while the process instances are still being executed. However, this *dynamic workflow evolution* [12] is not a trivial issue due to the fact that the termination of running process instances could result in the loss of process history, sensitive data, time, and eventually customers [8]. This is particularly unnecessary when a change has to be applied to a single process instance only. In some long-running processes, further steps depend on the current results, but these steps can't be always foreseen and have to be added or modified during the process execution.

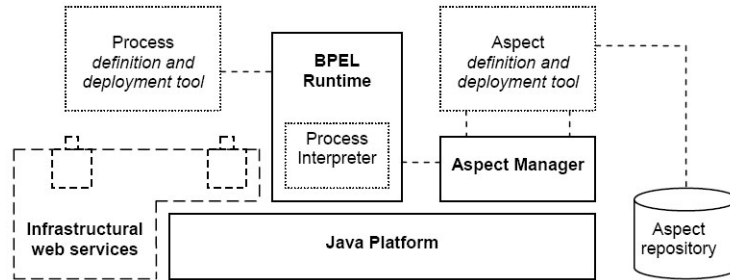
### 4.3 Approaches

In this section we are going to present several approaches to increase adaptability for BPEL processes using the aspect-oriented programming paradigm. The presented approaches allow, in particular, process adaptability during run-time.

**AO4BPEL.** AO4BPEL represents an aspect-oriented extension to BPEL and was introduced by A. Charfi et al. in [3]. It allows the definition of aspects using its BPEL extension during modeling as well as the deployment and activation during run-time. In this approach, every BPEL activity can be a possible

joinpoint. Since BPEL is an XML-based language, the authors propose to use the XML query language XPath for pointcuts to select joinpoints, even across process boundaries. In general, advices are implemented as BPEL activities fulfilling the crosscutting functionality, but sometimes the advice logic cannot be expressed in BPEL. In this case, the authors argue to call a *Java code execution web service*, as a so-called *infrastructural web service*, which invokes external Java methods similarly to the Java Reflection. The weaving process is realized through modifications of the BPEL activity interpretation flow.

The proposed BPEL engine was extended by several components and consists, therefore, out of (1) the BPEL run-time, which is an extended process interpreter that takes also aspects into account, (2) the Aspect Manager, which manages the aspect execution, (3) the aspect definition and deployment tool, which manages the deployment and activation of aspects, (4) the process definition and deployment tool and (5) the infrastructural web service.



**Fig. 5.** Architecture of an aspect-aware web service composition system as introduced for AO4BPEL in [3]

**Aspect weaving BPEL engine.** C. Courbis et al., the authors of the aspect weaving BPEL engine [4], focus their efforts on adapting and extending a BPEL engine through the application of AOP. Possible joinpoints in this approach are any BPEL instructions or process variable modifications. The proposed engine has a build-in dynamic aspect weaver and distinguishes between two different types of aspects, *engine* and *process aspects*. Both kinds of aspects can be weaved-in dynamically during run-time. Like in AO4BPEL, the pointcuts are defined using XPath for both types of aspects.

The engine aspects enhance the BPEL engine by adding new features to it, like tracing and debugging, and can be only added before or after a joinpoint. The corresponding advices are written in Java and registered in the engine aspect repository. The BPEL interpreter realizes the execution of engine aspects at a joinpoint through the implementation of the visitor design pattern. Hereby, the engine checks if an advice is to be executed before and after the execution of a visit method.

Process aspects, on the other hand, extend a process model or concrete process instance by additional functionality, and offer the options before, after, replace or delete. The process aspects, on the other hand, are written in a custom notation, using XPath for pointcuts and BPEL for advices, and are woven into a process using transformations on the abstract syntax tree (AST) of the BPEL process. These transformations can only be applied at certain points during execution to ensure the integrity of the system.

As opposed to AO4BPEL, this approach also offers the definition of a selection policy for the used WSs in a process. At run-time, according to the defined policy, the engine can select from an UDDI registry, a WS that complies the signature and the constraints. Hereby, the authors argue, a process can be adapted to improve the performance or quality of service.

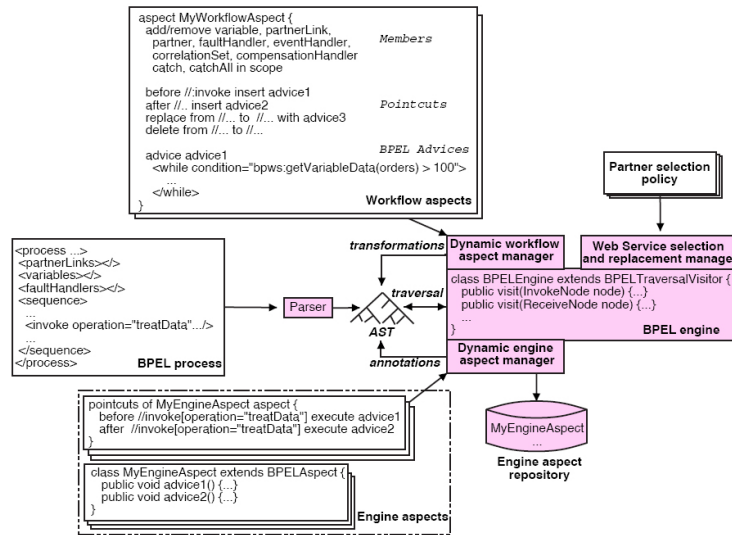


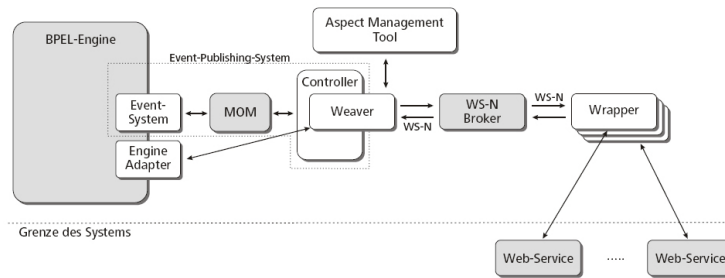
Fig. 6. Architecture of an aspect-weaving BPEL engine as in [4]

**BPEL'n'Aspects.** BPEL'n'Aspects as introduced in [12] and [10] tries a more generic approach to enhance process adaptability with AOP. The authors didn't extend BPEL, but used standard WS technologies allowing the application of the AOP paradigm not only to BPEL but also to ordinary WS technology as well. In their work, they propose a generic extension of a BPEL engine, for publishing navigation and lifecycle events, and making use of the publish/subscribe paradigm.

In this approach, joinpoints are specific BPEL elements, like activities, transition conditions, etc. To address these joinpoints, the pointcuts are realized through WS-Policy attachments representing subscriptions to navigation events

from the engine. The advices in this approach are realized as WS operations on concrete ports. The aspects, in BPEL'n'Aspects, are represented by WS-Policies and consist out of subscriptions to events and WS operations.

The event-system of the BPEL engine publishes notifications of certain process life cycle events, like the invocation of a BPEL activity using WS-Notifications. A message-oriented middleware (MOM) filters out the interesting messages and publishes them to the weaver. The weaver acts in this approach as a broker, receiving the event notifications from the engine and publishing these to subscribing wrappers using WS-Notification. The wrappers act as a gateway and realize the WS operation invocations representing the advices. A wrapper publishes the results of the invocation to the subscribing weaver, which signals these back to the engine. An aspect management tool handles the deployment of aspects at run-time. Furthermore, BPEL'n'Aspects offers *composite aspects*, a group of aspects that can be applied either completely or not at all. If aspects belong together, this functionality avoids that some of the aspects are already being executed while the rest is still being defined.



**Fig. 7.** BPEL'n'Aspects: BPEL engine architecture with Event-Publishing-System as in [12]

#### 4.4 Overview

In comparison to the other presented approaches, BPEL'n'Aspects is the most generic. Unlike AO4BPEL, this approach doesn't extend the business process language, but uses standard WS technologies instead. It also doesn't dictate a specific BPEL engine architecture but offers a non-intrusive extension of the BPEL engine. It supports dynamic weaving of aspects during run-time, especially on a per-instance basis. Through its generic approach, BPEL'n'Aspects and the presented infrastructure present a solution to process adaptability that allows the use with any legacy BPEL processes and environments, yet, not limiting it to BPEL exclusively, but offering its application to any WS compliant infrastructure. All presented approaches support adaptability with respect to changes of control and dataflow, only the aspect weaving BPEL engine also supports changes of WS instances.

	AO4BEL	Aspect weaving BPEL engine	BPEL'n'Aspects
Joinpoints	BPEL activities	Any BPEL instruction or process variable modification	Specific BPEL language elements, like activities, transition conditions, etc.
Pointcuts	XPath	XPath	Subscription to navigations event from engine (WS-Policy attachments)
Advices	BPEL activities or Java method calls: before/after/instead	<ul style="list-style-type: none"> <li>• Engine aspects: Java code: before/after</li> <li>• Process aspects: BPEL: before/after replace/delete</li> </ul>	WS operations at concrete ports: before/after/instead
Aspects	Custom BPEL extension	Custom notation	WS-Policy
Weaving	Modification of interpretation flow	<ul style="list-style-type: none"> <li>• Engine aspects: triggers on annotations in process AST</li> <li>• Process aspects: transformations on process AST</li> </ul>	Observing and signaling of navigation events to wrappers
Adaptation support	Change of control flow, data flow	Change of control flow, data flow, WS-instance	Change of control flow, data flow

Fig. 8. Overview of the presented approaches

## 5 Conclusion

We presented in this paper what the aspect-oriented programming paradigm can do. The concept is relatively simple, and the goal is clear: to enhance modularity and reuse. Therefore, AOP has a high potential of adding flexibility to software and services. But not only software and services, in this case programming in the small, can profit from the paradigm, but concepts like processes, representing programming in the large, as well. We could show how adaptability is a strongly needed feature in systems that work with processes.

We could show where web services lack flexibility and also miss their goal of interoperability and loose coupling used in a service-oriented environment. An approach was presented, which can fill this gap through the application of the AOP paradigm. The web service management layer adds loose coupling to applications on the client side offering the possibility of dynamically interchangeable web services depending on selection and management aspects. Overall, AOP can increase the flexibility of services in many different ways and its application offers a great gain.

Web service flows, like BPEL processes, inherit the deficiencies of the web service technology and are too inflexible in reaction to ever-changing business rules. This paper showed that there are several approaches how we can add adaptability to WS-flows. These approaches use the concepts of the AOP paradigm to achieve this goal. The approach of AO4BPEL extends BPEL and defines advices in BPEL itself and is, therefore, too restrictive. The aspect-weaving BPEL engine, on the other hand, has its focus on the special architecture of a BPEL engine with two kinds of aspects in different languages and is a too intrusive approach. The last presented approach, BPEL'n'Aspects, is the most generic, using well established standards like the WS-Policy and BPEL to define aspects

and giving an extension to a BPEL engine, which can be easily adapted to others. Also, this approach isn't limited to BPEL but can be adapted by any WS compliant infrastructure and can be used with legacy infrastructure, thereby, preserving investments.

The aspect-oriented programming paradigm by Kiczales can have a huge impact on the flexibility of software, as well as services and processes, and we could show a definite increase of agility.

## References

1. Baligand, F., Monfort, V.: A Concrete Solution for Web Services Adaptability Using Policies and Aspects. In Proceedings of ICSOC '04, New York, NY, USA, December 2004.
2. Böhm, O.: Aspektorientierte Programmierung mit AspectJ, Heidelberg, dpunkt Verlag, 2005, ISBN: 3-89864-330-1
3. Charfi, A., Mezini M.: Aspect-Oriented Web Service Composition with AO4BPEL. In Proceedings of ECOWS '04, Erfurt, Germany, September, 2004.
4. Courbis C., Finkelstein A.: Towards an Aspect-Weaving BPEL-engine. ACP4IS Workshop, 3rd AOSD conference, Lancaster, UK, 2004.
5. Dijkstra, E.: On the role of scientific thought. Selected writings on Computing: A Personal Perspective. New York, NY, USA: Springer-Verlag New York, Inc., pp. 60–66.
6. Forgác, M., Kollár, J.: Static and Dynamic Approaches to Weaving. 5th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence and Informatics. Poprad, Slovakia, 2007
7. Gradecki, J., Lesiecki, N.: Mastering AspectJ: Aspect-Oriented Programming in Java. New York, NY, USA: John Wiley & Sons, Inc., p. 4, 2003, ISBN: 978-0-471-43104-6.
8. Karastoyanova, D., Buchmann, A.: Extending Web Service Flow Models to Provide for Adaptability. In Proceedings of OOPSLA '04 Workshop on "Best Practices and Methodologies in Service-oriented Architectures: Paving the Way to Web-services Success", Vancouver, Canada, 2004.
9. Karastoyanova, D., et al.: Extending BPEL for Run Time Adaptability. In Proceedings of 9th International Enterprise Distributed Object Computing Conference (EDOC 2005), Enschede, Netherlands, 2005.
10. Karastoyanova, D., Leymann, F.: BPEL'n'Aspects: Chiming into Orchestrations, 2007 - to appear.
11. Kiczales, G., et al.: Aspect-Oriented Programming. In Proceedings of ECOOP'97, Finland, 1997.
12. Schroth, R.: Konzeption und Entwicklung einer AOP-fähigen BPEL Engine und eines Aspect-Weavers für BPEL Prozesse. Thesis No. 2523, University of Stuttgart, Germany, 2006.
13. Verheecke, B., Cibrán, M.: AOP for Dynamic Configuration and Management of Web Services. International Conference on Web Services Europe 2003, Erfurt, Germany, 2003.
14. Wikipedia: Aspect-Oriented Programming: [http://en.wikipedia.org/wiki/Aspect-oriented\\_programming](http://en.wikipedia.org/wiki/Aspect-oriented_programming)