Adopting Aspect-Oriented Software Development in Business Application Engineering[*]

Christoph Pohl, Anis Charfi, Wasif Gilani, Steffen Göbel, Birgit Grammel, Henrik Lochmann, Andreas Rummler, Axel Spriestersbach
SAP Research (CEC Karlsruhe, Darmstadt, Dresden, Belfast)
<firstname.lastname>@sap.com

ABSTRACT

SAP is the world's largest vendor of enterprise software. SAP Research is interested in understanding, evaluating, and applying aspect-oriented techniques in the context of large enterprise systems. This interest is also reflected by our involvement several European and national research projects on Aspect-Oriented Software Development (AOSD). We report on existing aspect-oriented concepts at SAP and present a case study that illustrates several non-obvious crosscutting concerns in business software. We also discuss the benefits and challenges that arise when applying AOSD to large scale industrial projects and present a road map to adopting AOSD at SAP for productive use.

1. MOTIVATION

Aspect-oriented concepts have recently gained significant attention in the research community. But large-scale industry adoption is rather sluggish. SAP, the world's largest vendor of enterprise software, investigates reasons for that in this application. The goal of this paper is to analyse how aspect-oriented concepts could help SAP to tackle hard development problems and which roadblocks could prevent this adoption.

Section 2 gives a brief overview of the state of the art in AOSD. The first core contribution of this paper is a survey of comparable concepts in SAP's ABAP [22] environment in section 3. Section 4 introduces our simplified case study from the domain of enterprise software. As a contribution to the community, it demonstrates non-obvious crosscutting concerns in enterprise applications. It is followed by a general comparison and evaluation of AOSD against conventional software development practices in section 5 and a roadmap assessing the productive adoption of AOSD concepts at SAP in section 6. Finally, we conclude in section 7 with a summary and recommendations.

[*]This work is partially supported by the EC-funded projects AMPLE, MODELPLEX and VIDE, and the German BMBF/DLR-funded project feasiPLe.

2. STATE OF THE ART

Evolutionary development of programming languages has always been marked by greater abstraction and separation of concerns. However, the prevailing paradigms of object-orientation and imperative programming have their disadvantages in this respect. For instance, the code implementing a certain concept tends to be scattered across application code. We discuss typical examples in section 2.3.

Aspect-Oriented Programming (AOP) [24] is a paradigm that mitigates this problem by introducing concepts to improve the encapsulation of crosscutting concerns as separate aspect modules. Aspect-orientation has gradually influenced not only implementation techniques but also preceding development stages, i.e., architecture modelling, design, and even requirements engineering [18]. These different streams are nowadays subsumed under the term *Aspect-Oriented Software Development (AOSD)* [12].

2.1 Aspect-Oriented Programming

AOP introduces four main concepts to improve the modularity of crosscutting concerns. *Join points* are well-defined points in the execution of a program where some crosscutting functionality should be executed. Examples of join points in an object-oriented code include method calls and field access. A *pointcut* is a construct to select a set of related join points, e.g., all calls to public methods that return a string. An *advice* is a piece of crosscutting functionality, which is associated with a pointcut. An advice can be executed before, after, or instead of the join points that are selected by the associated pointcut. An aspect is a module that encapsulates the implementation of a crosscutting concern. In addition to pointcuts and advices, an aspect may have fields and methods. It can also define introductions, i.e., extensions of existing classes in terms of additional members.

An aspect-oriented application consists of two components: the *base code*, i.e., the classes that implement the core business logic of the application, and the *aspects*, i.e., the modules that implement the crosscutting concerns. These two components are composed with a so-called *aspect weaver*. One difference between the existing AOP approaches is the weaving time, i.e., when the injection of the additional behavior encapsulated by aspects is integrated with the functional components of the system. In static weaving approaches, this process is carried out at compile-time and in dynamic weaving approaches it is carried out at load-time or at runtime.

Several aspect-oriented languages are available such as AspectC++ [34], AspectC [15], JBossAOP [21], and CaesarJ

[5]. AspectJ [24], which is an aspect-oriented extension of Java, is currently the most mature AOP language and it provides good tool support.

2.2 Aspect-Oriented Modelling

Model-Driven Software Development (MDSD) [35] motivates the lifting of fine-grained code structures to coarser-grained models. In contrast to model-based development, which just uses models for illustrative purposes, MDSD uses Domain-Specific Languages (DSL) as metalevel abstractions to iteratively refine system design, and to configure the generation and assembly of artefacts. This abstraction process shall approximate the developers mental model of the underlying implementation and, hence, reduce the necessary effort to produce them. Based on developed models for a system architecture, appropriate code generators produce later on executable runtime code.

Aspect-Oriented Modelling (AOM) [3, 1] incorporates the integration of aspects in model-driven development in order to provide a better modularisation of concerns already above fine-grained code structures. Models may modularise features as aspects, and artefacts may represent advices. From this viewpoint, the distinction between merging models and weaving aspects gets blurred.

Well comparable to the common AOP approach on code level, Aspect-Oriented Modelling combines at least two different models, where the *core model* represents the implementation basis an *aspect model* is to be woven into. A weaving description contains the necessary information to identify core model parts, which shall be extended by an aspect model.

Probably one of the most popular approaches to AOM is the Atlas Model Weaver (AMW) [2]. In that work, the relations between two or more models are expressed with a dedicated weaving model, which later serves as basis for the generation of transformation rules that define the weaving process.

Additional approaches to aspect-oriented modelling can be found in [1]. Most of these approaches provide constructs for modularising concerns and others for specifying concern composition. The approaches differ in various regards such as the targeted models (i.e., structural or behavioural models), the level of abstraction, the level of concern separation, the modelling constructs (whether they extend UML or use custom notations to model aspects), the language independence (i.e., whether they are bound to specific aspect languages), etc.

2.3 Use Cases for AOSD

A number of well-known – mostly technical – crosscutting concerns have been reported to be solvable with aspect technology. For instance, an assessment of the Apache Tomcat web server in [24] revealed both crosscutting and modular concerns, such as, Logging, tracing, XML parsing and URL pattern matching. While *logging* and *tracing* are evident examples of crosscutting concerns, the latter two are well modularized. Other classical examples are *persistence* [31] and *transactions* [21]. A whole chapter in [12] has been dedicated to improving *security* by modularising, e.g., access control or input checking, with AOSD techniques. Another field of research deals with *concurrence* and *synchronisation* as aspects [10]. For instance, request handling in web servers, event handling in GUIs, or debugging, are use cases where AOP can coordinate between concurrent aspects and base applications. Last not least, specialised aspect languages can be used for explicit *distributed* programming [28, 5, 27]. This brief list is not exhaustive. It primarily illustrates the breadth of AOSD use cases.

2.4 Industry Adoption of AOSD

Successful applications of AOSD in industrial projects have been reported, for instance, by Motorola [9], Siemens [37], or HP [26], mostly for typical AOP use cases as described in section 2.3. Duck [11] reports about the most frequent industrial applications for aspects: enforcement, development process, exploration and logic as well as infrastructure aspects. Most projects used Aspect-Orientation at programming level. More recent work from Motorola combined AO with Model-Driven Software Development [9].

Industrial application domains include VLSI tool development [26] and distributed telecommunication infrastructure software [9]. For enterprise software, application of aspects is mostly limited to typical "technial" aspects like monitoring [16] or middleware containers like JBossAOP [21]. However, no case studies are known that use AOSD for developing crosscutting business process logic in large enterprise application platforms.

2.5 Established Alternatives

Obviously, industry has tackled the outlined issues before in various ways. A number of traditional techniques based on, for example, design patterns, templates/pre-processors, copy/paste, patching/instrumentation, etc., have been employed long before the emergence of AOP tools to deal with the crosscutting concerns. In [33, 8, 14], the authors demonstrated how they could handle crosscutting concerns with design patterns. The work in [38] showed how the crosscutting code is separated and woven at compile time back into the C++ code by instantiating the templates. Pre-processor statements are also employed in the C/C++ domain to enable/disable code implementing the crosscutting concerns. Patching/instrumentation [13] is popular in the open source community, to update/replace the code implementing the crosscutting concerns (such as fixing of security holes, etc.). Copy/paste is oldest and most cumbersome, time consuming, and error prone technique to implement the crosscutting concerns, such as logging, etc., by copying-adapting and then pasting the crosscutting code throughout the whole application in all the required methods.

Though effective to a limited degree for separating the crosscutting code, these techniques fail to offer mechanisms to modularise the crosscutting concerns, thereby compromising reusability. Additionally, there is no flexible way (pointcut mechanism) to describe the target join points. The absence of quantification support means that the application of an aspect to a large number of varying join points is complicated and demands excessive effort. Last, but not the least, the oblivious principle of AOP does not hold as the component code in most of these techniques, except patching, is aspect-aware.

3. EXISTING AO CONCEPTS AT SAP

Although SAP systems support a wealth of predefined business processes in its core, this core must be adapted to the needs of particular enterprises and industries. For instance, it is obvious that there are fundamental differences for purchase order processing in process industries like oil & gas compared to discrete industries like pharmaceuticals. This has led to the implementation of specialised enterprise extensions and industry solutions in the ABAP [22] language stack, SAP's proprietary language and programming environment for business processes[1]. Deploying such a solution at a customer site usually requires further adaptations. So-called *user exits* provide predefined hooks to plug in customer specific functionality. They have later been extended by *Business Add-ins (BAdIs)*, which rely on stable interfaces in the core system and the appropriate industry solution.

However, on one hand, BAdIs may still not cover all customer extension scenarios, and on the other hand, the large number of BAdI interfaces required from the core for implementing various industry solutions are increasingly harder to maintain across release upgrades. The first problem was tackled by allowing customers to implement *source code modifications* (cf. patching in section 2.5). A so-called modification assistant tracks these source code modifications and enables them in production systems. Obviously, source code modifications introduce a number of lifecycle and maintainability issues. For instance, source code modifications in an industry solution may change core functionality, which immediately causes conflicts upon core upgrades. Customer extensions on top of industry solutions complicate this situation even further. For these reasons a new approach for the management of modifications had to be developed. The outcome is the Enhancement Framework [23].

The Enhancement Framework has been introduced in SAP NetWeaver 2004s, Release 7.0, with the goal to unify possible types of modifications/enhancements in ABAP. At the core of the framework there is a simple structure consisting of a hook and an element that can be attached to this hook. The main function of the EF is the modification, replacement, and enhancement of objects that form the technical basis of a SAP system.

There are three elementary concepts in the EF for modifying/enhancing such objects, which are shown in Fig. 1.

Enhancement Options (EO) are defined as positions in repository objects (*points* A–E and *section* B), where enhancements can be made. Two types of EO exist: *explicit* and *implicit* options. An explicit EO can be currently defined by explicitly flagging points or sections in the source code of ABAP programs as being extensible by a developer, which are then enhanced by object plug-ins. Enhancement points resemble the concept of insertion, while sections are associated to the concept of replacement. There may be only one section active at a time, while several active points are possible.

[1]For other purposes, such as user interfaces and service composition, Java is widely used at SAP.

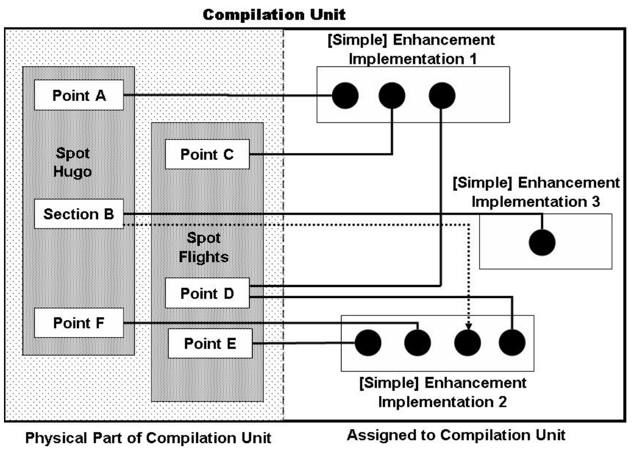

Compilation Unit
[Simple] Enhancement Implementation 1
Point A
Spot Hugo
Point C
[Simple] Enhancement Implementation 3
Section B
Spot Flights
Point D
Point F
Point E
[Simple] Enhancement Implementation 2
Physical Part of Compilation Unit
Assigned to Compilation Unit

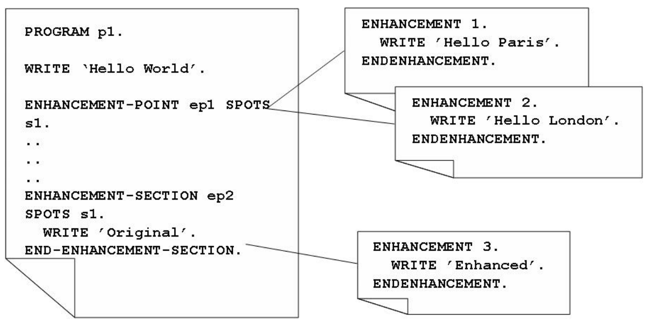
Figure 1: Enhancement Spots & Implementations

In contrast to explicit EOs, implicit EOs are special points in ABAP programs that can be enhanced (not shown in Fig. 1), i.e., at the end of all programs, after the last statement, at the beginning and end of subroutines, at the end of Function Modules or at the end of all visibility areas (public, protected, and private) of classes. These points can be enhanced by source code plug-ins, parameters or attributes. To that end, implicit EOs fulfil the AOP property of *obliviousness* [12].

In principle, EOs are managed by *Enhancement Spots (ES, "Hugo" and "Flights" in Fig. 1)* and filled by *Enhancement Implementations* (EI, 1–3), but there are exceptions. Implicit EOs always exist and do not require ES, but are also enhanced by EI. ES are used to manage explicit EOs and carry information about the positions at which EOs were created. One ES can manage several EOs of a repository object. Conversely, several ES can be assigned to one EO. Both ES and EI can be hierarchically composed (i.e., simple and composite ES/EI). ES are directly supported by the Enhancement Builder, an integral part of the ABAP Workbench, SAP's development environment for the ABAP language. EOs are created by creating ES element definitions. Such an option can be called by using ES element calls. Both concepts combined form an EO definition. An ES element definition must be assigned to at least one ES. A simple ES is assigned to exactly one enhancement technology (source code or BAdI), while composite spots are used for semantic grouping. For this reason it is possible to group semantically related EOs under one simple ES.

Enhancement Implementations (EI) are the counterpart for ES. At runtime one or more EI can be assigned to a single ES (e.g., Point D is implemented EI 1 & 2 and Section B in EI 2 & 3). There are several types of EI: Source Code Enhancements, Function Module Enhancements and Global Class Enhancements. Source Code Enhancements represent the direct insertion of source code at predefined locations in ABAP programs (cf. Fig. 2). These locations can be defined by implicit and explicit EOs. Function Module Enhancements represent the enhancement of parameter interfaces. For example a new optional parameter can be added to the interface of a function module. In addition via Global Class Enhancements new attributes can be added to repository objects or special pre-/post-methods can be realised, which are called directly before/after ABAP methods.

The introduced concepts can be roughly compared to concepts of Aspect-Oriented Programming: *Enhancement Options* resemble *Join Points* and *Enhancement Implementations* compare to *Advices*. An example is shown in Fig 2. In this example a simple program is extended by several enhancement implementations. Enhancement 1 is inserted at the position marked with ENHANCEMENT-POINT and can optionally be overwritten by Enhancement 2. In contrast Enhancement 3 is not inserted at some particular point, but replaces a section marked with ENHANCEMENT-SEC-



```
PROGRAM p1.

WRITE `Hello World'.

ENHANCEMENT-POINT ep1 SPOTS
s1.
..
..
..
ENHANCEMENT-SECTION ep2
SPOTS s1.
  WRITE 'Original'.
END-ENHANCEMENT-SECTION.
```

```
ENHANCEMENT 1.
  WRITE 'Hello Paris'.
ENDENHANCEMENT.
```

```
ENHANCEMENT 2.
  WRITE 'Hello London'.
ENDENHANCEMENT.
```

```
ENHANCEMENT 3.
  WRITE 'Enhanced'.
ENDENHANCEMENT.
```

Figure 2: Enhanced ABAP source code

TION.

One may assume that *Enhancement Spots* resemble the concept of *Pointcuts*, but this is actually not the case. Pointcuts and the associated pointcut language offer a way to quantify a set of relevant Join Points out of all existing Join Points. An analogy in the EF does not exist as such. The assignment of EI to EO via ES is done manually case-by-case. The activation is performed using the Switch Framework, a business configuration tool. Hence, the second AOP property of *quantification* [12] is not completely fulfilled by the Enhancement Framework, at least not in its current version.

In general it is recommended to use BAdIs whenever it is possible, because their well defined interfaces give core providers a firm control over possible enhancements. Second, explicit EOs should be preferred over implicit EOs for the same reason. Source code enhancements (i.e., patching) should be the last option.

4. CASE STUDY: SALES SCENARIO

One of the major challenges when motivating the application of AOSD technologies is the illustration of their potential usefulness. Common examples are often restricted to the integration of rather simple aspects, such as logging or security, into existing applications (cf. section 2.3). On the other hand, industrial Software Product Line Engineering (SPLE) [36] uses well known traditional techniques to exclude or integrate features into deployable end user products (cf. section 2.5).

In this section we present a simplified industry-scale software product line to demonstrate non-obvious crosscutting concerns. We propose that AOSD technologies are one means to implement variability in such product lines and, hence, may alleviate the challenges industry-scale SPLE is facing. We will start with a general overview about the business domain, continue with an architectural overview and conclude with the presentation of above-mentioned crosscutting concerns.

4.1 Introduction to the Domain

Our case study demonstrates business application engineering in the domain of enterprise software. This domain is rather large. Exemplary solutions centered around Enterprise Resource Planing (ERP) include Product Life Cycle Management (PLM), Supply Chain Management (SCM), or Supplier Relationship Management (SRM). Such solutions must be adapted and customized to a particular company (no two companies are the same), business applications often have thousands of configuration settings. To reduce the complexity for the sake of conciseness, we focus on one specific sub-domain in this paper – Customer Relationship Management (CRM) – combined with some parts of the aforementioned solutions. The case study is not intended to be complete. It concentrates on relevant parts in a variability-driven context with a significant amount of cross-cutting functionality. Therefore, the presented example is called *Sales Scenario* to clearly distinguish from CRM as explained by Buck-Emden and Zencke in [6].

Main purpose of the Sales Scenario is the holistic management of business data, including central storage and access controlled retrieval. It focuses on product sales processes. Such processes comprise opportunity management, quotations to customers, sales order and invoice processing. In the following, the intention of these components is explained in more detail.

Opportunity Management.

Opportunity management is a pre-sales process that supports sales personnel in actively tracking potential selling possibilities. It provides a structured approach to turning customer opportunities into sales contracts.

The opportunity process starts by identifying and creating an opportunity, e.g., with an address after a sales contact at a fair. Then, additional prospect attributes are gradually added, such as customer's budget and success probability. It is evaluated and qualified, i.e., feasibility is clarified, customer information is gathered, and a selling team is defined. If a go decision is made, a quotation is created, as explained in the next section. After that the opportunity should be closed and the reasons for success or failure entered.

Quotation Management.

Successful opportunities result in creating quotations (i.e., offers) using quotation management functionalities. Accordingly, a quotation template is configured with details of the given sales opportunity, including prospect details and the products to be offered.

Based on categorisation of the prospect in a customer group, estimated sales volume, and sales probability, resulting in an overall customer rating, an individual pricing strategy is used to calculate a discount for the customer. Once the quotation is complete, it is sent to the potential customer by e-mail or letter. In case of positive response on the quotation, an order is created.

Order Management.

After the sales office has contacted the customer and received positive feedback to a quotation, it is converted into an order. Creditworthiness of the customer is checked during sales processing by interacting with a payment module. By interacting with warehouse management, a so-called "availability to promise" check ensures required capacities and sufficient stock in the warehouse. In case of multiple stocks only those warehouses sufficiently close to the shipping address are included. After all necessary steps are performed, the order is triggered to be shipped to the customer. The explicit order's state is set accordingly.

Payment Processing.

If payment is to be integrated into the process, it would be activated automatically upon creating a binding sales order.

Depending on the method of payment offered by the system and selected by the customer, an automatic debit transfer from the customer's account can be triggered or an invoicing document can be attached to the delivery.

The order status is set to "sent" by an employee as soon as the order is delivered to the customer. Once an open invoice is paid by the customer, the order is marked as paid and may be closed. In case that an open or already completed sales order is returned by the customer, an approval process will be triggered, which involves the sales management for commitment.

4.2 Architecture Overview

The high-level architecture is depicted in Fig. 3 in FMC notation [20]. It outlines the key entities and their interactions/interfaces according to the functionality described above.

Prominent are well modularisable, coherent features on the one hand and wide spread, crosscutting features on the other. Exemplary for coherent features are *Product Management* and *Payment Processing* components. Such components are to a certain degree self-contained and mainly expose their functionality for use by other components. In the AOSD-centric context of this paper, such features are not that interesting. Instead, the second feature type labels all those components that are scattered over a platform and interact actively with other components. Thus, such components encapsulate non-obvious cross-cutting concerns by widely distributing parts of their functionality. A detailed description of such features is given in the next section.

4.3 Non-obvious Crosscutting Concerns

4.3.1 Partner determination

Partner determination refers to the system ability to automatically find and complete partner information such as addresses in certain transactions and documents. That is, the user manually enters one or more partners, e.g., in an *Opportunity*, a *Quotation*, or a *Sales Order*, and the system determines and completes other partners and information by using several sources of information such as the business partner master data, the company organizational data, documents related to the current document, etc.

Figure 4 shows an example that illustrates how partner

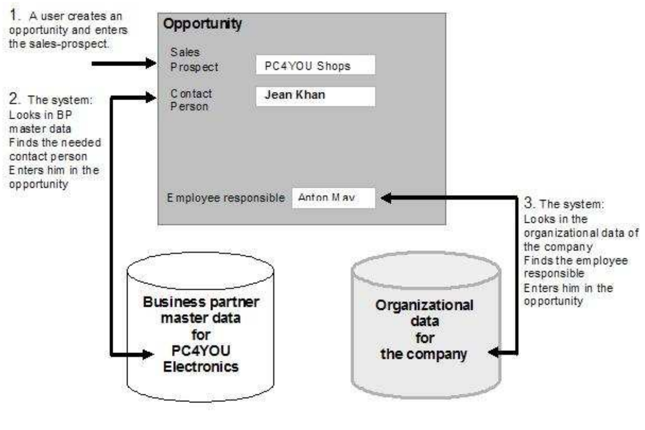

1. A user creates an opportunity and enters the sales-prospect.
Opportunity
Sales Prospect
PC4YOU Shops
Contact Person
Jean Khan
2. The system: Looks in BP master data Finds the needed contact person Enters him in the opportunity
Employee responsible
Anton May
3. The system: Looks in the organizational data of the company Finds the employee responsible Enters him in the opportunity
Business partner master data for PC4YOU Electronics
Organizational data for the company

Figure 4: Partner Determination Example

determination works. The user creates an *Opportunity* and enters the name of the sales prospect and the system enters the name of the contact person (by checking the partner master data), the address of the sales prospect, and the name of the responsible employee for this opportunity (using the company organizational data).

The way partner determination is done can be very different depending on the business process, the business transaction, the partner functions in the transaction, and the companies that run the CRM software. Customers that use SAP CRM solutions can set up rules defining what data sources to use for each partner function (e.g., contact person, sold-to-party, ship-to-party, etc.) and in what order these sources are searched (so-called access sequences). They can also configure the time when partner determination is performed, e.g., when data is entered by the user or when the data is saved. Partner determination procedures bring together partner functions and access sequences.

Partner determination is a crosscutting concern as the respective code is scattered across several classes of the user interface as well as the business object classes of the CRM application. Partner determination may be triggered in the UI classes, e.g., when a user enters the sales prospect for an opportunity and also in some business object classes such as *Opportunity* and *Sales Order*.

4.3.2 Consistency checks

Several consistency checks have to be performed when the state of the opportunity object or some of the associated objects changes. These consistency checks are crosscutting because they affect multiple classes, i.e., the same checks need to be performed when attributes of objects that are defined in different classes change. Consequently, the code that enforces them is scattered across the implementation of several classes.

We classify the consistency rules into two types according to the degree of crosscutting:

- *Simple constraints* involve only one business object. For example, the constraints C0, C1, and C2 define an *Opportunity* as being inconsistent if, e.g., one of the following conditions is true:

  C0 Opportunity.processStatusValidSinceDate > current_date

  C1 SalesForecast.expectedProcessingPeriod.EndDate is not set

  C2 SalesForecast.expectedProcessingPeriod.EndDate < SalesForecast.expectedProcessingPeriod.StartDate

- *Complex constraints* involve more than one object and consequently their enforcement in an object-oriented design is scattered across at least two classes. For example, the constraints C3 and C4 are complex constraints, which should be fulfilled to guarantee that an *Opportunity* is consistent. The constraint C3 specifies that the value of the opportunity attribute *processStatusValidSince* should be smaller than start date of the *expectProcessingPeriod* attribute of the associated *SalesForecast* object. C4 constrains the attributes *phaseProcessingStartDate* and *expectedProcessingStartDate* of respectively the *SalesCycle* and *SalesForecast* objects that are associated to an opportunity:

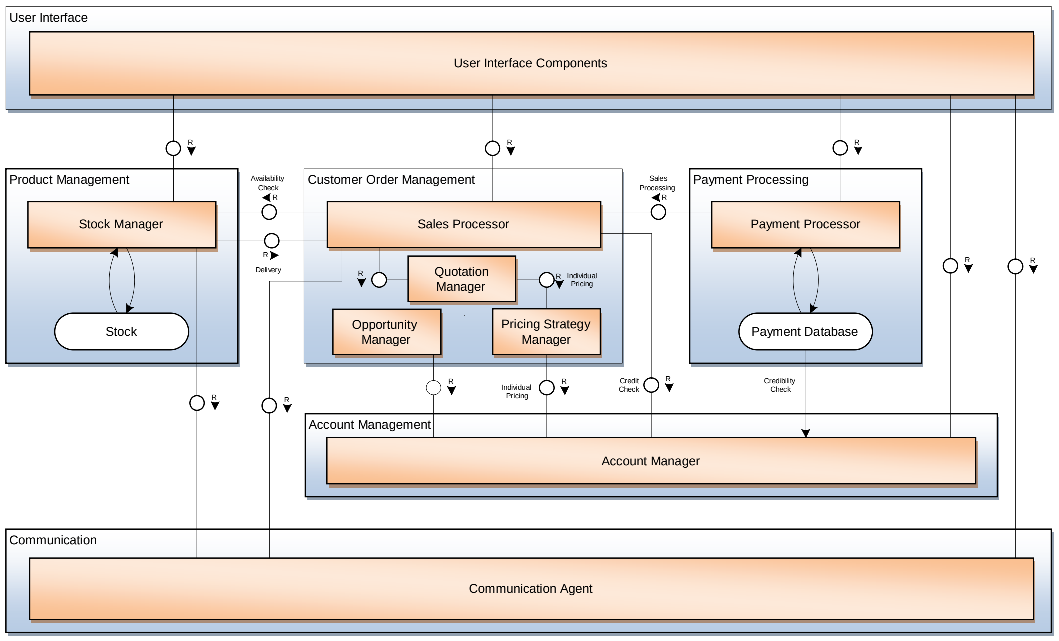


Figure 3: Sales Scenario Architecture Overview

C3 Opportunity.processStatusValidSinceDate < SalesForecast.expectedProcessingPeriod.StartDate

C4 SalesCycle.phaseProcessingPeriod.StartDate < SalesForecast.expectedProcessingPeriod.StartDate

Appropriate logic is needed to check these consistency constraints and prevent their violation. This logic should be triggered when the fields corresponding to the constraints are modified and also when the setter methods of these fields are called. For instance, to enforce the complex constraint C3, appropriate logic is required in the method *setProcessStatusValidSinceDate* of the class *Opportunity* to ensure that the date parameter is smaller than *expectedProcessingPeriod.StartDate* in the associated *SalesForecast* object. Also similar logic is required whenever the field *ProcessStatusValidSinceDate* is set directly without calling the setter method. Moreover, some logic for checking C3 is needed in the method *setExpectedProcessingPeriod* to verify that the *StartDate* of the period parameter is bigger than the value of the attribute *processStatusValidSince* in the associated *Opportunity* object. Similar logic is needed when the respective field is set directly.

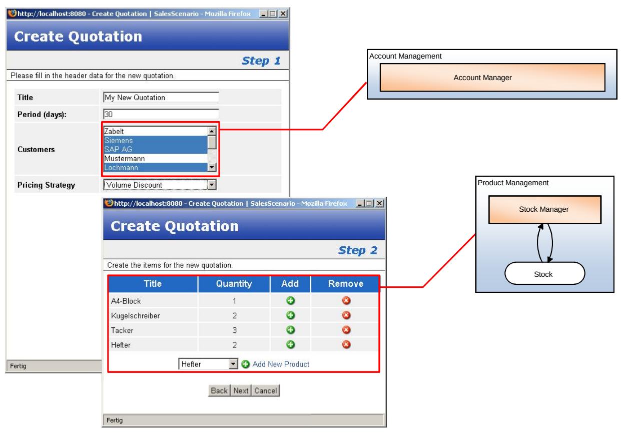
4.3.3 *General interdependencies among components*

Software applications are commonly composed of several architectural parts, such as database, workflow and process models, or the user interface. Similarly, the implementation of meaningful features contains combined contributions to at least some or rather all of these parts. Even if a feature contributes new functionality to only one architectural part, it may interact with others indirectly due to interdependencies. For instance, the quotation management as feature of the *SalesScenario* product line, had to be bundled with appropriate parts providing necessary data structures for persistence and in memory storage, UI parts allowing the user to manipulate quotations, as well as process descriptions, e.g., to define step-wise workflows for creating and editing quotations. While the different parts themselves depend on each other (a concrete quotation attribute may have a certain representation in the user interface), they may also interact with architectural parts of other components. To clarify of this, consider the following examples:

A *quotation* data object consists of *header information*, such as a title, creation date etc., *several prospects*, i.e., potential customers, which shall receive the quotation and *several products* that are proposed to the prospects. This illustrates a dependency of the quotation management component to account and product management components. Without these components, the dedicated handling of customers and products could not be included in a quotation, leading to different user interface dialogues and data structures for the quotation management.

Figure 5 illustrates this relationship. Depicted are the first two pages of the "Create Quotation" wizard that facilitates the stepwise creation of quotation entities. As shown, each page collects data for the quotation to create. In this respect, the first page interacts with the account management component to retrieve all available business partners while the second page queries the product management for all registered products (cf. Fig. 3). If these two components were not available due to a certain product configuration, the shown quotation management wizard had to be altered.

A *customer order* data object depends very similarly on account and product management, because an order holds mainly the same information like quotations. Additionally, there are dependencies to the quotation management, recalling the automatic order creation out of an existing quotation (cf. Sec. 4.1). To illustrate another interaction with an implementation of payment, imagine triggering a payment process according to a certain sales order state. All these


http://localhost:8080 - Create Quotation | SalesScenario - Mozilla Firefox

Create Quotation

Step 1

Please fill in the header data for the new quotation.

Title	My New Quotation
Period (days):	30
Customers	Zabelt, Siemens, SAP AG, Mustermann, Lochmann
Pricing Strategy	Volume Discount

Fertig

http://localhost:8080 - Create Quotation | SalesScenario - Mozilla Firefox

Create Quotation

Step 2

Create the items for the new quotation.

Title	Quantity	Add	Remove
A4-Block	1	⊕	⊗
Kugelschreiber	2	⊕	⊗
Tacker	3	⊕	⊗
Hefter	2	⊕	⊗

Hefter — Add New Product

Back | Next | Cancel

Fertig

Figure 5: Create quotation wizard pages with integrated functionality of foreign components

interdependencies are scattered over the mentioned architectural parts.

Hence, components contain contributions to different architecture parts depending on each other (one concern → different places in implementation). Different components contain interdependencies between each other (removing one component necessitates removal of wide-spread minor assets in other components). Aspect-oriented programming techniques are one mean to cluster the different contributions in a modular and manageable way.

4.3.4 *Process-related Concerns*

In large software development organisations such as SAP, it is becoming increasingly hard to consider the ever increasing amount of development guidelines and governance processes. For instance, sanitation of all external system input is required to prevent cross-site scripting attacks; similarly, access control checks are mandatory to prevent unprivileged data retrieval. It is obviously easy to miss some of the code spots where these aspects need to be applied, when programming them manually. On the other hand, many of these guidelines could be expressed by aspects. This can help to improve quality and correctness of delivered modules as well as maintainability of base code due to the cleaner separation of concerns.

Another example for such process-related concerns is the challenge of industry solutions on top of a core platform as mentioned in section 3. Industry variants of business processes are typically highly crosscutting. Every single variation of an industry solution may affect multiple process steps in a process chain as well as multiple application layers. For instance, a simple field extension of a business object like "business partner" has impacts on almost every process working on business partner data as well as on every architectural layer from database up to the user interface. A modular handling of such multilayer adaptations would greatly increase maintainability and thus, also total cost of ownership (TCO).

5. ASSESSMENT OF AOSD CHALLENGES

As reported in various related works (see section 2), a well directed application of AOP principles in the development of systems leads to high degrees of modularity, reusability, flexibility, granularity, adaptability, evolution, extensibility, etc. However, a number of challenges have to be taken into account when evaluating the use of AOP and/or AOSD in practical projects.

Maintainability.

Aspects are usually tightly coupled to the base code they are advising. That means any changes to the base code (e.g., renaming methods, etc.) may potentially invalidate existing aspects or even activate unintended join points. These implicit dependencies require special governance during development. In turn, if providers of base code want to guarantee upwards compatibility of aspectual extensions, they are essentially condemned to rigidly keeping their poincut structure as it is. There are three main research directions addressing the so called aspect-base coupling problem. The first direction [4, 17, 19] of research involves the counteraction of aspect expressiveness, while giving up part of the code *obliviousness*. The second approach investigates alternative ways of modular reasoning in the presence of aspects, as in [25]. A third direction of research focuses on the definition of semantically advanced pointcuts at a higher level of abstraction in terms of the program semantics [29] rather than program syntax. However, such pointcuts are still not supported in the AOP languages and tools that are mature enough to be used in industry.

Scalability.

While the above mentioned maintenance problems may be bearable in small projects, remedies do not scale to large Software Supply Chains as they can be found in the domain of Enterprise Software. If a cascade of platform provider(s), independent software vendors (ISV), integrators, and customers build hierarchical extensions depending on each other, more governance is required. The SAP ERP solution is a good example for such Software Supply Chains: SAP builds its own extensions and industry solutions on top; ISVs provide additional extensions, and customers further extend, customize and configure the deployment. If one uses aspects as extensibility mechanism in such a context one would have thousands and even more possible configurations that are no longer manageable. Without appropriate concepts and tools the overall view is lost. Although there is some research on the dependencies and interactions between multiple aspects [32], these concepts do not scale to complex software supply chains where multiple parties are involved.

Legal Issues.

Aspects could become a security risk because of them having comprehensive control over all join points in the system. Such an exposure of even sensitive parts of system to aspect code for intrusion is not acceptable for instance in the area of business software because that software already implements several legal regulations, e.g., for tax calculations and compliance with laws such as SOX and Basel II. If everyone is allowed to modify every part of the application the application provider can no longer guarantee that his application is consistent and that it complies with the legal laws and regulations. In such context, advanced concepts for encapsulation/accessibility are required to make sensitive join points selectively unavailable for pointcuts.

Distribution.

AOP has focused so far mostly on local execution environments. However, most business applications are distributed especially in the case of large enterprise software. This property raises several requirements such as supporting distributed join point models, remote advice execution, distributed activation and deactivation of aspects, etc. Although there are some research efforts in these areas [28, 5, 27], they are still at the stage of research prototypes that cannot be used in an industrial environment. The AspectJ language supports only quantification over local execution.

Learning Effort.

AO raises the level of abstraction and increases modularity similarly to Object-Orientation (OO). However, to really gain the benefits promised by AOSD, the adopters (e.g., developers at SAP) need to understand and learn AOSD concepts and the underlying principles such as separation of concerns, quantification, obliviousness, etc. This can be a challenge for developers. Without addressing this issue, developer may improperly use AOSD. In many companies, developers are using object-oriented languages but they still code as if were are using a procedural language. The same applies also for AOM: A proper understanding of MDSD [35] processes is necessary. Guiding tool support can help to address this challenge.

6. TOWARDS ADOPTING AOSD AT SAP

While AOP has been applied to a couple of industrial projects it did not reach the expected adoption level in large-scale enterprise software development projects inside and outside SAP. Reasons for this are often a *"combination of technical, social, psychological and commercial considerations [that] need to be addressed"* [7]. Many of those adoption barriers have been reported in literature, but were not applied to SAP development processes and technologies. Therefore this section discusses those concerns in the context of SAP needs.

6.1 Social and psychological barriers

New software development paradigms need to be beneficial either for the overall software product or individual work, otherwise they will not be accepted and used by developers. Successful applications of AOP in development projects are a good way of demonstrating and communicating potential benefits of AOP for SAP. The case study introduced in section 4 was created with this intention. However, this is only the first step. Learnings from this case study need to be promoted actively and applied further to actual development projects. Cross-fertilisation between the major language stacks of Java and ABAP [22] seems promising. Existing ABAP enhancement concepts described in section 3 could be extended with advanced AOP concepts, such as a generic point-cut language for quantifying join points.

The lack of comparable success stories is also a result of missing experience and understanding of AOP concepts. Surveys in internal SAP newsgroups yielded almost no feedback. It seems that AOP concepts are little known within SAP. Therefore it is often not considered as an option.

The level of seniority reported to be required for successfully adopting AOP [37, 39] makes the situation worse. Although it is possible to apply AOP at the abstraction level of procedural languages like C [15], the basis for applying full-fledged AO is a thorough understanding of Object-Orientation in design and implementation of base code. This cannot be taken for granted since especially ABAP does not mandate object-oriented programming. While developing advices and specifying pointcuts by quantifying join points is a complex task that requires a comprehensive domain and platform know-how, programming base code becomes easier because of the cleaner separation of concerns. Aspect-Oriented Development is thus not necessarily better suited for small teams. It is only important to staff aspect developer positions with senior personnel like development architects.

Developers often get the impression of loosing of control when AOP is introduced, because arbitrarily invasive aspect code may be applied to "their" base code. Their concern is that they might be made responsible for errors they actually cannot influence. This is an AOP challenge in general that could largely be resolved by appropriate tools, which in turn is a technical adoption barrier (see below). For instance, the IDE could present eligible join points to the developer for approval. This way, the value of aspects becomes more obvious than with weavers working behind the scenes, especially for tedious concerns such as product standards and development guidelines. As an example, the Eclipse plug-in for AspectJ [24] provides the means to visualise the activated join points of some aspect in given base code.

Recommendations.

- Gather experiences with advanced (mostly Java-based) AO concepts using small case studies (cf. section 4) and apply lessons learned to the ABAP Enhancement Framework (cf. section 3).

- Actively promote lessons learned and educate potential adopters.

- Gradually prove the value of Aspect Orientation in the form of guiding tool support instead of enforcing obscure aspect weaving.

6.2 Technical adoption barriers

Many technical challenges of AOSD were described in section 5. We have seen above that the *non-obvious* code modification by aspect weavers and the additional level of indirection are a major concern for developers. For understanding the overall interactions between the concerns at runtime, powerful tool support for tracing, verifying, debugging at the same abstraction level is of utmost importance [37]. While many external tools are available for Java (mostly for AspectJ [24], tool support in the ABAP stack [22] is limited to the functionality of the Enhancement Framework (see section 3). Tooling for potential AO extensions of ABAP would need to be integrated in this environment.

A Forrester report observes that *"AOP is an extension to object-oriented programming [for issues] that OOP does not address well [. . .]"* and *"most AOP projects [. . .] avoid modifying the Java language and runtime (JVM). Ultimately, a better approach may simply be to fix the shortcomings of Java."* [39]. Consequentially, the full control over ABAP gives SAP the opportunity to integrate language extensions more easily. The introduced Enhancement Framework is an ideal starting point for that.

Aspect-orientation at the modelling level, i.e., modular reasoning about cross-cutting features at earlier development phases, requires a model-driven development process infrastructure [35] to meaningfully leverage its potential. SAP already develops, for instance, user interfaces in a model-driven process. But development of business logic is largely model-based, i.e., models are mainly used for illustrative and documentary purposes, and media breaks between lifecycle stages need to be bridged manually. Consequentially, introducing AOM to SAP will work only in concert with adopting MDSD.

Another technical barrier of more fundamental nature is routed in the aspect-base coupling problem also described in section 5. The resulting maintainability problems and the broken encapsulation of objects are even aggravated in staged development environments, where layered functional extensions are developed by independent stakeholders. SAP's business suite represents the base platform of such a software supply chain, involving ISVs, partners, and customers. In [30] an approach is proposed to make use of explicit aspect interfaces to counteract the invasive nature of aspects. The underlying idea is to define a more explicit contract between base code and aspects, by allowing only certain joint points to be targeted as points for advice invocation. The pointcut description as such stays the same, while the base code is re-processed to take care of potential aspect-oriented extensions. The ABAP Enhancement Framework described in section 3 tackles this problem by explicitly recommending the use of implicit enhancement options (the nearest equivalent to "real" aspects) *only* for those rare cases where explicit enhancement options and regular BAdIs (comparable to framework hooks) are insufficient.

Recommendations.

- Tool support for debugging, tracing, and refactoring is key for wide adoption of AOP. Otherwise, the benefits of higher abstraction are quickly frustrated.
- Heavy-weight language extensions are often more expressive than non-invasive pre-processor instructions. They should be considered an option, especially in the case of ABAP.
- Adopt model-driven development processes, not only to increase productivity, but also to prepare the ground for aspect-oriented modelling.
- More flexible mechanisms for avoiding aspect-base coupling are needed, especially in complex software supply chains.

6.3 Economical considerations

Forecasts about the potential cost savings of applying AOP as an alternative to traditional methods (e.g., AspectJ in Java projects) are difficult. The return on investment of AOP platform extensions, like the Enhancement Framework in ABAP, is even harder to estimate. Forrester suggests to *"Carefully monitor the defect rate and the time needed to fix systems that use aspects. This is the only way to determine whether aspects are costing you more than they are worth"* [39]. This means you have to constantly compare these values with historical data for traditional development methods. For investments on the ABAP side, experiences with existing enterprise, industry, and customer extensions based on (implicit) enhancement options should be analysed in that way.

The same report also notes the risks of aspect-base coupling and feature (aspect) interaction: *"In some implementations, the introduction – or removal – of an aspect can break existing code in unanticipated ways"* [39], which becomes even more dangerous when third parties are involved in software supply chains (see above). Strict governance should be applied to all existing extension points and aspects to avoid side effects.

Recommendations.

- Assess existing extensions using the SAP's Enhancement Framework with respect to defect rates and problem resolution times.
- Gather experiences with AOP in internal Java projects to evaluate alternative approaches before applying it to mission critical products.
- Avoid promoting implicit enhancement options (i.e., join points) as external interfaces. Provide explicit interfaces where possible.

7. SUMMARY

In this paper we have presented novel insights to applying AOSD to the business software domain. We have surveyed existing aspect-oriented concepts in the SAP-proprietary ABAP language stack. A real-world case study was used to demonstrate non-obvious, domain-specific crosscutting concerns as new application scenarios for the AOSD community. Finally, we presented a roadmap to overcoming identified challenges of AOSD adoption in the context of SAP product development.

The conclusions and recommendations can be summarised as follows: Adoption of aspect-oriented concepts at SAP should be driven primarily through small internal Java projects, where a wide range of external approaches is already available. The Enhancement Framework is the ideal starting point for adoption on the ABAP stack. On both sides, AOP should not be promoted as an extension mechanism for ISV/partner/customer development because of maintenance issues (aspect-base coupling). Guided development with support for compliance with coding standards, business rules, etc. can help to gradually lower reservations against weaving behind the scenes. Aspect-Oriented Modelling may help to raise the abstraction level but model-driven development processes need to be in place to leverage its full potential.

The authors will continue to work on the identified challenges for driving the adoption of AOSD concepts in strategic business areas.

8. REFERENCES

[1] AOSD-Europe Network of Excellence Deliverable 11: Survey of Analysis and Design Approaches. `http://www.aosd-europe.net/deliverables/d11.pdf`.

[2] Atlas Model Weaver. ATLAS Group, INRIA. `http://eclipse.org/gmt/amw/`, 2007.

[3] O. Aldawud. A UML Profile for Aspect Oriented Programming. In *Workshop on AOP at OOPSLA*, October 2003.

[4] J. Aldrich. Open modules: Modular reasoning about advice. In *19th ECOOP*, pages 144–168. Springer, 2005.

[5] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. Overview of CaesarJ. *Transactions on AOSD I*, 2006.

[6] R. Buck-Emden and P. Zencke. *mySAP CRM: The Official Guidebook to SAP CRM 4.0*. Galileo Press, 2004.

[7] A. Colyer. Towards Widespread Adoption of AOSD, 2003.

[8] C. A. Constantinides, A. Bader, T. H. Elrad, P. Netinant, and M. E. Fayad. Designing an aspect-oriented framework in an object-oriented environment. *ACM Computing Surveys*, 32, 2000.

[9] T. Cottenier, A. van den Berg, and T. Elrad. The Motorola WEAVR: Model Weaving in a Large Industrial Context. In *AOSD*, 2007.

[10] R. Douence, D. L. Botlan, J. Noye, and M. Suedholt. Concurrent aspects. In *GPCE*, Oct. 2006.

[11] A. Duck. Implementation of AOP in non-academic projects. In *AOSD*, 2006.

[12] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development.* Addison-Wesley, Boston, 2005.

[13] M. E. Fiuczynski, R. Grimm, Y. Coady, and D. Walker. patch (1) considered harmful. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, 2005.

[14] B. Goetz. Java theory and practice: Decorating with dynamic proxies. `http://www.ibm.com/developerworks/library/j-jtp08305.html`.

[15] M. Gong, V. Muthusamy, and H.-A. Jacobsen. The Aspect-oriented C compiler. `http://research.msrg.utoronto.ca/ACC/WebHome`.

[16] K. Govindraj, S. Narayanan, B. Thomas, P. Nair, and S. Peeru. On using AOP for Application Performance Management. In *AOSD*, 2006.

[17] W. Griswold, M. Shonle, K. Sullivan, Song, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, pages 51–60, Jan./Feb. 2006.

[18] J. Grundy. Aspect-oriented requirements engineering for component-based software systems. In *ISRE*, pages 84–91. IEEE Computer Society, 1999.

[19] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD*, pages 60–69. ACM Press, 2003.

[20] Intervista AG Deutschland. Functional Modeling Concepts - FMC. `http://www.fmc-modeling.org/notation_reference`, 2007.

[21] JBoss AOP - Aspect-Oriented Framework for Java. `http://labs.jboss.com/jbossaop/`.

[22] H. Keller and S. Krüger. *ABAP Objects: ABAP Programming in SAP NetWeaver*. Galileo Press, 2nd edition, 2007.

[23] K. Kessler. A New and Improved Approach to SAP Industry Solutions: How the Switch and Enhancement Framework Now Consolidates SAP Industry Solutions with the ERP Core. *SAP Professional Journal*, 3:29–44, 2006.

[24] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[25] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE*, pages 49–58. ACM Press, 2005.

[26] M. Mortensen and S. Ghosh. Using Aspects with Object-Oriented Frameworks. In *AOSD*, 2006.

[27] L. D. B. Navarro, M. Südholt, W. Vanderperren, B. D. Fraine, and D. Suvée. Explicitly distributed aop using awed. In *AOSD*, pages 51–62. ACM, 2006.

[28] M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut: a language construct for distributed aop. In *AOSD*, pages 7–15. ACM, 2004.

[29] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *ECOOP*, pages 214–240. Springer, 2005.

[30] C. Pohl, A. Rummler, and B. Grammel. Improving traceability in model-driven development of business applications. In *ECMDA Traceability Workshop*, pages 7–15. ECMDA, 2007.

[31] A. Rashid and R. Chitchyan. Persistence as an aspect. In *AOSD*, pages 120–129. ACM Press, 2003.

[32] F. Sanen, E. Truyen, W. Joosen, A. Jackson, A. Nedos, S. Clarke, N. Loughran, and A. Rashid. Classifying and documenting aspect interactions. In *ACP4IS workshop at AOSD*, 2006.

[33] C. Schwanninger, E. Wuchner, and M. Kircher. Encapsulating cross-cutting concerns in system software. In *ACP4IS workshop at AOSD*, 2004.

[34] O. Spinczyk, D. Lohmann, and M. Urban. AspectC++: an AOP extension for C++. *Software Developer's Journal*, 2005.

[35] T. Stahl and M. Völter. *Model Driven Software Development.* John Wiley & Sons, 2006.

[36] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process.* Addison-Wesley, 1999.

[37] D. Wiese, R. Meunier, and U. Hohenstein. How to Convince Industry of AOP. In *AOSD*, 2007.

[38] Z. Yao, Q. L. Zheng, and G.-L. Chen. AOP++: A generic aspect oriented programming framework in C++. In *GPCE*, 2005.

[39] C. Zetie. Aspect-oriented programming considered harmful. Technical report, Forrester, 2005.