SPECIFIC TARGETED RESEARCH PROJECT

INFORMATION SOCIETY TECHNOLOGIES

FP6-IST-2005-033606

VIsualize all moDel drivEn programming
VIDE

WP 3	**Deliverable number D.3.2** **Specification of the AOC to be Supported by VIDE** **(Report + Demonstrator)**

Project name:	Visualize all model driven programming
Start date of the project:	01 July 2006
Duration of the project:	30 months
Project coordinator:	Polish - Japanese Institute of Information Technology
Work package Leader:	Fraunhofer FIRST
Due date of deliverable:	31 August 2007
Actual submission date	10 October 2007
Status	developed / draft / **final**

Project supported by the European Commission within Sixth Framework Programme

Document type:	Report + Demonstrator
Document acronym:	D3.2
Editor(s)	Anis Charfi, Jaroslav Svacina
Reviewer(s)	*Joachim Hänsel, Piotr Habela*
Accepting	Kazimierz Subieta
Location	www.vide-ist.eu
Version	1.0
Dissemination level	**PU**/PP/RE/CO

Abstract

The purpose of work package 3 is to investigate strategies for the integration of aspect oriented composition techniques in model driven development and make recommendations to the design and implementation work packages on the most suitable approach for supporting aspect-oriented composition in VIDE. In Deliverable 3.1, the state of the art in aspect-oriented modelling was introduced as well as examples of crosscutting concerns in a typical SAP business application. Moreover, a demonstrator on aspect-oriented composition at the PIM level was presented. In the current deliverable, we will evaluate the selected modelling and composition techniques and present a specification of aspect-oriented composition in VIDE including the corresponding profiles, textual and visual syntax, and the model transformations. The main elements of this specification will be illustrated by means of two aspects: consistency checks and partner determination.

The VIDE consortium:

Polish-Japanese Institute of Information Technology (PJIIT)	**Coordinator**	**Poland**
Rodan Systems S.A.	Partner	Poland
Institute for Information Systems at the German Research Center for Artificial Intelligence	Partner	Germany
Fraunhofer	Partner	Germany
Bournemouth University	Partner	United Kingdom
SOFTEAM	Partner	France
TNM Software GmbH	Partner	Germany
SAP AG	Partner	Germany
ALTEC	Partner	Greece

Executive Summary

The VIDE project aims at developing *"a fully visual toolset to be used both by IT-specialists and individuals with little or no IT-experience, such as specific domain experts, users and testers."*[1]. Therefore VIDE investigates "*visual user interfaces, executable model programming, action- and query-language-semantics, AOP and quality assurance on the platform-independent modelling level, service oriented architecture (especially Web services integration) and business process modelling."*. VIDE is aimed to be embedded in the Model Driven Architecture of the OMG, thus supporting modelling both on a domain-oriented computation-independent layer (CIM), a platform-independent layer (PIM), and generating models on a platform-specific layer (PSM). VIDE is primarily targeting the domain of business application software.

The goal of Work Package 3 in the VIDE project is to investigate integration strategies for adding advanced aspect-oriented software composition in the platform-independent modelling phase of MDD processes. The resulting knowledge allows integrating the aspect-oriented modelling and composition techniques into the VIDE language and architecture. The benefit for the VIDE project will be shown by evaluating the developed concepts and by assessing the used technology.

In this work package we have researched aspect orientation on the PIM level using Customer Relationship Management business scenarios that are provided by SAP. The lack of support in object-oriented modelling techniques for modularizing crosscutting concerns in the provided scenarios raised the need for aspect-oriented techniques while modelling business processes and business applications.

Our research included the evaluation of different existing approaches in the domain of aspect oriented programming by applying them to the relevant phases of Model Driven Development as well as the investigation of existing approaches in the area of aspect-oriented modelling.

Based on the research results a suitable concept for modelling aspect-oriented constructs, such as aspect, advice, and pointcut was developed. To ensure a straightforward integration of these constructs into the VIDE metamodel we have selected the UML Profile extension mechanism.

To allow the VIDE model compiler to deal with the aspect-oriented modelling concepts that we have developed, we present an aspect composition strategy, which is based on model-to-model transformations. The feasibility of the developed concepts and strategies was shown by a proof-of-concept prototype, which consists of UML Profiles for aspect modelling and two transformations respectively for join point matching and aspect weaving at the model level.

Deliverable 3.1 presented the state of the art in aspect oriented composition at the model level and provided an analysis of the chances and risks for the investigated modelling and composition techniques. It also aimed at providing the required knowledge for integrating aspect orientation into the context of VIDE.

Deliverable 3.2 evaluates the proposed approach and gives a specification of the aspect oriented composition to be supported by VIDE.

[1] From the VIDE project summary in the Technical Annex I

Table of Contents

1 Introduction and Overview

The purpose of work package 3 is to research the combination of Aspect-Oriented Software Development (AOSD) and Model-Driven Software Development (MDSD). In fact, model-driven development can gain a lot from the modularization concepts that are introduced by aspect-orientation especially when the application behaviour is modelled. In that case, which is targeted in VIDE, modelling the behaviour becomes very similar to programming in a typical object-oriented programming language. As a result of this similarity, the benefits of AOSD can be very likely brought to behavioural modelling at the PIM level.

More precisely, work package 3 aims at identifying crosscutting concerns in data-intensive business applications and providing aspect-oriented constructs in VIDE to support a better modularization of these concerns. Such constructs will provide several benefits such as an easier understanding and maintenance of the application models, more reuse of the behaviour models, easier extensibility and customization, etc. In addition, work package 3 investigates appropriate ways to model and represent aspect-oriented constructs and to integrate them in the VIDE language. Another major contribution of this work package is to research the composition of the aspect models with the base VIDE models.

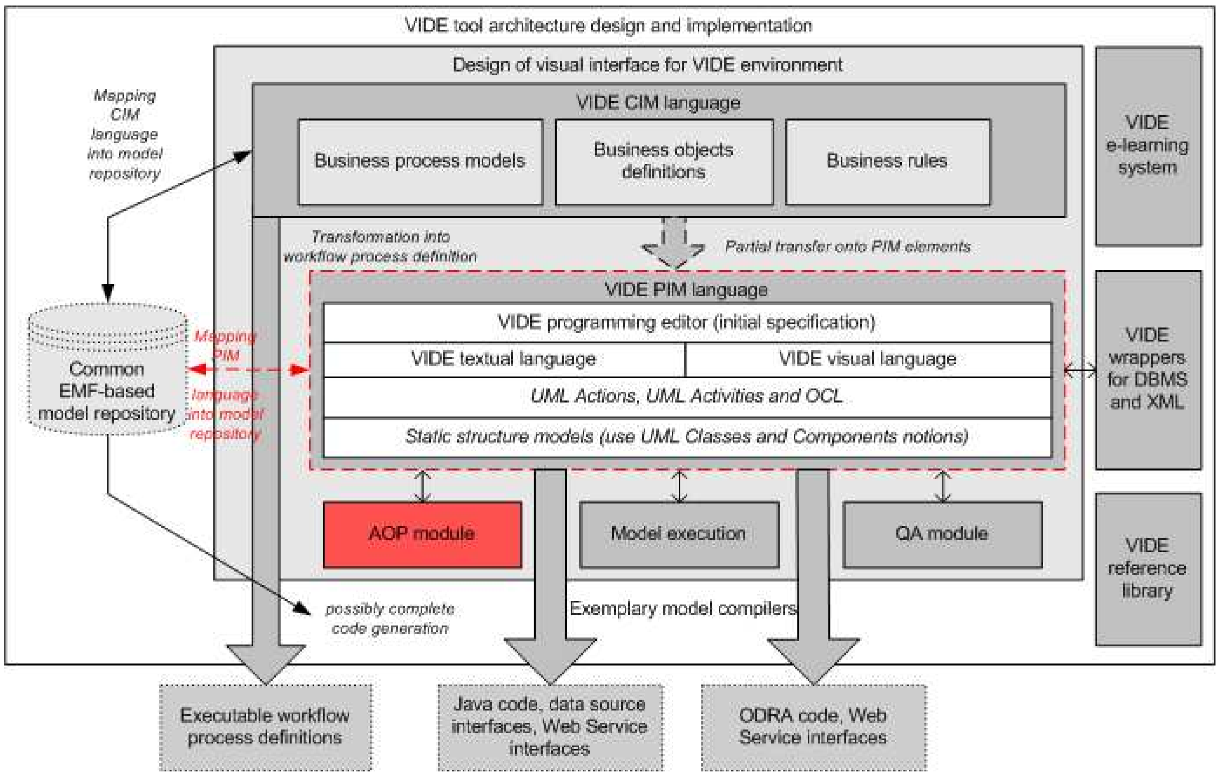

VIDE tool architecture design and implementation
Design of visual interface for VIDE environment
VIDE CIM language
Business process models
Business objects definitions
Business rules
VIDE e-learning system
Mapping CIM language into model repository
Transformation into workflow process definition
Partial transfer onto PIM elements
VIDE PIM language
VIDE programming editor (initial specification)
VIDE textual language
VIDE visual language
UML Actions, UML Activities and OCL
Static structure models (use UML Classes and Components notions)
Common EMF-based model repository
Mapping PIM language into model repository
VIDE wrappers for DBMS and XML
AOP module
Model execution
QA module
VIDE reference library
possibly complete code generation
Exemplary model compilers
Executable workflow process definitions
Java code, data source interfaces, Web Service interfaces
ODRA code, Web Service interfaces

Figure 0: WP3 in the VIDE context

The aim of work package 3 is to define aspect-oriented constructs in VIDE including the respective abstract and concrete syntax extensions together with the mapping to a model repository module. The output of this work package is the definition of the aspect-orientation module in the VIDE architecture, which is highlighted in Figure 0.

The current document is the second and last deliverable of work package 3. In D3.1, we presented the state of the art in aspect-oriented software development and model-driven development with particular focus on aspect-oriented modelling. In addition, we presented a few examples of crosscutting concerns in a typical SAP business application and also explained the benefits that are expected from using aspects in modelling business applications. The main part of that deliverable was a detailed description of a proposal for aspect-oriented modelling at the PIM level and the presentation of a proof-of-concept demonstrator. Both parts serve as foundation for the specification of aspect-oriented composition in VIDE, which will be presented in the current document.

1.1 Challenges

We identified three main challenges when integrating aspect-oriented constructs in the executable modelling language VIDE.

First, one has to choose the right application scenarios, which are valuable for VIDE partners and users. As VIDE targets especially data-intensive business applications, we used the opportunity management part of a SAP CRM software as business scenario. Although there are several interesting development aspects such as testing and monitoring, we focused rather on non-intuitive production aspects such as consistency checks and partner determination because most development aspects were already addressed in other research works.

The second challenge is the integration of aspect-oriented concepts into an executable modelling language at the PIM level. This is especially challenging as the VIDE meta model unifies UML actions and OCL expressions. Consequently, a thorough investigation of such integration is required. One has to explore ways to define, model and represent aspect-oriented constructs, such as advice and pointcut. The integration of these constructs has to be done in a non-invasive way using UML profiles. In addition, one has to define a powerful pointcut and advice language that can cope with the requirements of business applications w.r.t. crosscutting concerns and which is also well-adapted to the base language.

Third, an appropriate composition mechanism is required for composing aspects with the VIDE models. For that purpose, we developed model-to-model transformations to perform pointcut matching and advice weaving. The proof-of-concept implementation of the composition mechanism does not only show the feasibility but also allows us to detect potential technological problems.

1.2 Tasks

These challenges correspond to the following tasks[2]. Task 3.1 was mostly addressed in D3.1 but it is also covered in the current deliverable, which contains an evaluation of the proposed aspect-oriented modelling approach. Task 3.2 was part of Deliverable 3.1 and Task 3.3 is the main focus of the current deliverable.

[2] From the VIDE WP3 description of work in the Technical Annex

1.2.1 Task 3.1: Practical evaluation of AO modelling and composition in MDA

A demonstrator (for the sole purpose of the evaluation report) utilizing techniques selected in task 1.3 will be developed, which will show the suitability of the technique, investigate the maturity of its AO modelling approach and spawn hidden risks in composing AO models particularly for data-intense business applications. To this end a metamodel for the assessment of most suitable AO approaches will be developed, regarding AO model extensions, aspect weaving level and complexity of MDA transformations. Several parts of a given business application will be analyzed in order to identify composition scenarios crucial for business applications. The most important composition scenarios will be designed and executed using the most reasonable composition technique. The results will be assessed considering the identified factors that are important to an MDA development process. A report will summarize the experiment's results, discuss the collected data and in particular recommend an AO modelling technique and design for integrating AO composition into the VIDE environment.

1.2.2 Task 3.2: Provision of a knowledge base for AO software composition in MDA processes

By structuring the empirical data of Task 3.1 a standard body of knowledge for best practices of AO modelling and composition techniques in MDA development processes with a focus on the business application domain will be initialized. It addresses the maturity of existing AOP approaches as well as integration issues. The evaluation of this body will take place by dissemination of research results and empirical evaluation by the research community, software companies and tool vendors.

1.2.3 Task 3.3: The specification of the Aspect-Oriented composition mechanisms to be supported by VIDE

Based on the analysis performed and in cooperation with VIDE language definition activities of WP2, the aspect-oriented composition mechanisms for VIDE will be specified. The specification will cover the respective semantics, notation and visual user interface elements.

1.3 VIDE language requirements

This subsection revisits the requirements collected during the state of the art analysis performed during the WP1 of the project (as described in D1.1 document [22]) and indicates, what of them are relevant to the scope of work of this WP and how they have been addressed. Moreover, this subsection describes the further elaboration of those requirements that has been performed in the course of WP3.

1.3.1 Requirements specified in the course of work package 1 work

We provide here a list of requirements with respect to the VIDE project, collected in the deliverable document D1.1 (see that document for a detailed description of these requirements) and indicate those found relevant for the WP3 scope. In the column “comment” we provide the relation of each requirement to the VIDE language, which is the subject of this deliverable document. For clarification, we denote which topics are subject of other work packages. We also sketch how WP3 will cover the relevant goals.

Requirement Number	**Name**	**Priority**	**Comment**
REQ – NonFunc 1	Accessibility at the CIM level	Should	Outside WP3 scope. Addressed by D7.1 and the CIM-to-PIM transition support functionality to be described in D5.1.
REQ – NonFunc 2	CIM level collaboration	May	Outside WP3 scope. Supporting this requirement will be considered in the course of D9.3 development.
REQ – NonFunc 3	On-line support for CIM/PIM users	Should	Outside WP3 scope. Addressed in D5.1 (in the area of CIM-PIM navigation).
REQ – NonFunc 4	Clear and unambiguous notation – VIDE should have clear, comprehensible and unambiguous semantic description suited to the users of the VIDE tools	Should	Mainly addressed by D2.1 and D7.1. The notation for the AO-specific constructs has been defined in section 4.4 of this document.
REQ – NonFunc 5	Model view saliency – VIDE models views must be user-oriented.	Should	Mostly outside the scope of WP3: addressed by the CIM and PIM languages design (D7.1 and D2.1) as well as by the GUI design developed in D5.1.
REQ – NonFunc 6	Appropriate textual/graphical fidelity – VIDE must provide appropriate textual and graphical modalities for its users.	Should	Mostly outside the scope of WP3: addressed by the CIM and PIM languages design (D7.1 and D2.1) as well as by the GUI design developed in D5.1.
REQ – NonFunc 7	Timely feedback and constraints	Should	Outside WP3 scope. Supporting the work of multiple users on a common model will be considered in the course of D8.1 and D9.1 development.
REQ – NonFunc 8	Runnable and testable VIDE prototypes	Should	For the specific area of WP3 a Demonstrator for early experimentation and to provide a proof of concept has been provided.
REQ – NonFunc 9	Scalability of proposed solution – the proposed solution must at least conceptually scale to enterprise level.	Must	In meeting these criteria the WP3 depends on the PIM language design specified in D2.1. None of the AO constructs introduced seems to affect the conceptual scalability. Aspect composition is implemented as a model-to-model transformation, i.e., just another transformation in the MDA approach of VIDE.
REQ – User 1	Flexibility and interoperability of VIDE language and tools - The VIDE language and tools SHOULD have flexibility and be interoperable with some existing tools.	Should	This is assured by compliance of the PIM-level language to the MDA standards and technologies including in particular the OMG UML 2.1 and OMG OCL 2.0 specification and a standard-compliant framework MDT implementing their metamodels. The way WP3 provides AO notions into it (lightweight extension using a UML profile) and limiting the AO constructs to the PIM level by the use of the *horizontal composition* approach (cf. Section 3.2.1) does not interrupt that compliance.

REQ – User 2	Reuse of UML Standard – end users are very sensitive to using standards. A key aspect is that the VIDE language reuses as much as possible the UML standard.	Should	Respected by the D2.1 and the WP3 constructs dependent on it – as explained above. Moreover, the WP 3 uses a UML profile to define the aspect-oriented constructs.
REQ – Semantics 1	Semantics of VIDE Inte rnal Communication – a precise description of the semantics is needed sufficient for internal communication purposes within implementation stakeholders in the development of the VIDE tool.	Should	Met by making the introduced WP3 notions compliant with the language definition and standard metamodels defined in D2.1 and described using analogous means.
REQ – Semantics 2	Simple VIDE semantics – after a first analysis it seems sufficient that the semantics of VIDE is described in natural language.	Should	Met by making the introduced WP3 notions compliant with the language definition and standard metamodels defined in D2.1 and described using analogous means.
REQ – Lang 1	Usage of UML2 Behaviour (“Action Semantics”) – VIDE should use the behavioural model elements of UML2 (earlier known as “UML Action Semantics”), unless proven insufficient.	Should	Addressed by D2.1 and the introduction of the WP3-specific notions in a way consistent with that language. The behavioural parts of the aspects are the advices and the methods. Both of them are modelled using action semantics.
REQ – Lang 2	Simplified UML meta-model – If it turns out that • the UML meta-model is unnecessarily complex in a way that it blocks the creation of a sensible concrete syntax (see remarks on ConditionalNode), • not all of the UML meta-model can be covered	May	The aspect-oriented constructs defined in WP3 have been introduced into the PIM level language using minimum number of terms and depending on a lightweight metamodel extension mechanism.

	• elements are missing which are located in another needed language (like OCL) it may be changed.		
REQ – Lang 3	User Language & Concepts – the VIDE language and VIDE tools presented to a certain user groups SHOULD employ the language that is understood by the user group.	Should	The WP3 provides the language with a small number of additional constructs that seem orthogonal to the syntactic variability for some constructs introduced by D2.1. The primary group of users dealing with the WP3 defined constructs are Analysts / VIDE programmers.
REQ – Lang 4	Compliance with Standards – VIDE should not compete with existing adopted modelling standards, especially those adopted by the OMG, such as UML or BPMN.	Should	Compliance with UML maintained – as explained under REQ – User 1.
REQ – Lang 5	Deviation from Standards – VIDE may deviate in parts from existing standards, if a standard-conformant way is provided as well and if there are good reasons with respect to the overall user requirements.	May	The very idea of introducing the AO notions into UML can be considered a deviation from a fully standard-compliant solution. Note however, that the impact has been limited to the inside of the PIM level – particularly, the model compilers are not affected by that extensions.
REQ – Lang 6	Modularisation and extensibility – it should be possible to replace parts of the language with different artefacts and add additional language constructs for special business specific patterns. This requires the language to be structured in modules.	Should	The aspect-oriented constructs defined in WP3 are modularized in an AO Profile (cf. Section 5.2)
REQ – Lang 7	Language for CIM, PIM, PSM modelling: 1) VIDE SHOULD support requirements	Should	Ad. 1. Outside WP3 scope. To be addressed in D7.1. Ad. 2. Outside WP3 scope. Addressed by D2.1. Ad. 3. Outside WP3 scope. To be addressed in D6.1.

	definition tasks and business process description with BPML 2) VIDE SHOULD adopt action semantics for the modelling of executable PIM models 3) VIDE SHOULD provide support for target PSM environments e.g. Java, C++, or SmallTalk; VIDE should provide platform implementation mappings in PIMs or CIMs.		
REQ – Tool 1	Usage of industrially adopted tools – VIDE must use industrially adopted meta-modelling standards where applicable.	Must	Compliance with UML and a standard-compliant implementation of its metamodel maintained – as explained under REQ – User 1.
REQ – Tool 2	Meta-modelling Framework – VIDE must use EMF as its modelling framework.	Must	Compliance with UML and a standard-compliant implementation of its metamodel maintained – as explained under REQ – User 1.
REQ – Tool 3	Meta-modelling Concepts – VIDE meta-models should be constructed to be compatible with MOF concepts.	Should	Compliance with UML and a standard-compliant implementation of its metamodel maintained – as explained under REQ – User 1.
REQ – Tool 4	M2M Transformation Technology	Should	Outside WP3 scope. To be addressed by D6.1. However, note that the demonstrator used ATL transformations for pointcut matching and weaving (cf. Section 5.3)
REQ – Tool 5	M2T Transformation Technology	Should	Outside WP3 scope. To be addressed by D6.1.
REQ – Tool 6	T2M Transformation Technology	Should	Outside WP3 scope. To be addressed in D9.3.
REQ – Tool 7	Meta-modelling Framework	Should	Outside WP3 scope. To be addressed in D9.1 and D9.3.

REQ – Tool 8	Use of OCL – VIDE should re-use existing standards as UML (REQ – User 1), and in particular OC;. the goal is to achieve a seamless integration with the concrete syntax of the action language to be developed.	Should	Satisfied by D2.1 and the WP3 notions dependent on it.
REQ – Tool 9	CIM modelling standards.	May	Outside WP3 scope. To be addressed in D7.1.
REQ – Tool 10	PIM, PSM modelling standards – VIDE SHOULD provide support for PIM modelling with UML and action semantics; the meta-modelling standard for VIDE should be Ecore. VIDE SHOULD support well known PSM modelling standards (e.g. XMI for model and meta-model interchange, JMI for Java based PSM).	Should	Satisfied by D2.1 and the WP3 notions dependent on it.
REQ – Tool 11	Framework for CIM, PIM, PSM modelling	Should	Met. The way AO notions have been introduced does not limit the applicability of the frameworks being considered.
REQ – Tool 12	VIDE extensibility	Should	Outside WP3 scope. To be addressed by D9.3.
REQ – Tool 13	Integration and metadata interchange – VIDE should provide model and meta-data interchange capability by adopting the XMI standard.	Should	Met by D2.1 and not affected by WP3 work.
REQ – Tool 14	Model driven approach The VIDE tool strictly follows a model driven approach as stipulated in figure 9 page 120 of the D.1.1 deliverable	Must	The design of VIDE language depends on the OMG four level meta-modelling architecture and is compliant with the approach mentioned. The AO notions have been encapsulated into the PIM level; hence their impact onto the overall approach is limited.

Table 1: Summary of the relevant requirements identified during the WP1 work

1.3.2 Further elaboration of the requirements in the course of work package 3

Modularity at the model level is the main requirement behind the work performed in WP3. In other words, the aim of the work package is to enable modular VIDE models especially with respect to crosscutting concerns. In fact, modularity at the model level would bring several benefits with respect to understandability, maintainability, extensibility, etc.

For that purpose, we evaluated aspect-oriented software development techniques with respect to their suitability for modelling business applications. We did the investigation from two perspectives: the AO language (i.e., what AO constructs are needed in the VIDE context) and the composition mechanism (i.e., how to weave aspects with the base models). Thereby, we used industry-scale examples of crosscutting concerns that are found in a SAP CRM application such as consistency checks and partner determination.

Then, we developed a proposal for the integration of aspect-oriented modelling constructs in the VIDE language using UML Profiles. In addition, we compared and evaluated several composition approaches and finally opted for horizontal composition, which was implemented in the proof-of-concept demonstrator by two groups of model-to-model transformations (respectively for pointcut matching and aspect weaving).

To confirm our start assumptions on the benefits of aspect-oriented modelling, we compared the understandability and maintainability of PIM models with aspects (i.e., using the proposed AO constructs) and without aspects (using object-oriented PIM modelling).

1.4 Document Outline

After giving an overview of this deliverable in *Section 1*, we will evaluate in *Section 2* the selected composition and modelling techniques that were presented in D3.1 by discussing several variations of our aspect-oriented modelling approach. Then, we introduce two evaluation criteria and some quality factors and metrics in order to use them in the following section for comparing business application modelling with aspects (as proposed in VIDE) against the traditional object-oriented modelling.

In *Section 3,* we present in detail two crosscutting concerns in a typical SAP business application from the Customer Relationship Management context. Then, we show how the behaviour belonging to these concerns is modelled once without aspects and once with aspects. Thereby, we will use the criteria presented in Section 2 and the respective metrics to compare both alternatives and draw conclusions out of this comparison.

Section 4 presents the main contribution of this deliverable, which is a specification of aspect-oriented composition in VIDE. This section defines the profiles for aspect-oriented modelling as well as the specifications of the necessary transformations for aspect composition. Moreover, it presents proposals for extensions to the textual and visual syntax of the VIDE language to integrate aspects.

Section 5 gives a summary of this deliverable and discusses open issues and problems. Moreover, it gives an outlook to the future.

2 Evaluation of Selected Composition and Modelling Techniques

The most pressing problem in software development seems to be complexity. Most target domains and projects get more and more complex. Tackling this complexity during software development needs new techniques and methodologies beside the currently used ones. Both Aspect-Oriented Software Development (AOSD) and Model-Driven-Development (MDD) provide new ways to confine and reduce complexity in creating solution domains and developing software. Both approaches try to solve the complexity problem with different but complementary ideas. So it seems natural to combine these approaches and reap the benefits of both for overcoming complexity in software development.

This section provides an evaluation of the selected AO modelling and AO composition approaches, which were described in D3.1. The latter will be reviewed in the following and different variations of aspect-oriented composition will be discussed and compared on different levels. The last subsection introduces several evaluation criteria and corresponding metrics for the empirical evaluation of the proposed AO modelling approach. The results of that evaluation are presented and discussed in Section 3.

2.1 Review of the Approach

After analyzing the provided business scenarios, the goal was the provision of a suitable approach for modelling crosscutting concerns and also for the composition of the modelled artefacts to a woven model.

In the first version of the aspect-oriented modelling approach, which was presented in D3.2, it was not possible to model the complete consistency check, but that version has shown the suitability for the domain. Due to the flexible design, the AO Profiles were extended to provide the required constructs for a suitable modelling of the identified consistency checks. The resulting Profiles are described in Section 4.

The chosen aspect composition supports the required binding kind (around) and the weaving concept is applicable to the selected scenarios. Moreover, the approach supports different instantiation strategies. To allow the realisation of different instantiation and weaving strategies, we decided to encapsulate the aspect behaviour in separate classes in the woven model. This gives us the flexibility to allow adding additional features with minimal effort. On the other side, this approach produces a lot of required infrastructure model elements and especially a lot of additional object creations and operation calls. This has two relevant effects. The readability and understandability of the woven model is decreased. This effect is not critical, because the woven model is not intended to be read by a human, but rather to be processed by a model compiler. On the other hand, the additional object creations and operation calls can have a negative impact on the performance of the generated software. An analysis of the impact on the performance of the woven model should be done in the future, but this goes beyond the scope of WP3 and is not considered.

The Demonstrator was intended to spawn risks of the chosen technology for the aspect-oriented composition. After the implementation of several model-to-model transformations using ATL [18], this technology seems to be suitable for the realisation of the aspect-oriented composition in VIDE. Nevertheless, the unstable version of ATL hinders the implementation of the demonstrator and causes a high effort.

2.2 Discussion of Variations

Aspect-oriented composition approaches can be realized by using different concepts on different levels. To give an overview of possible concepts and to show, where the developed approach is settled, the following sections describe and compare different variations of aspect-oriented composition.

2.2.1 Composition Layer Variations

The different variations of the aspect composition have already been particularly described in D3.1. Therefore, this section shortly summarizes the main results.

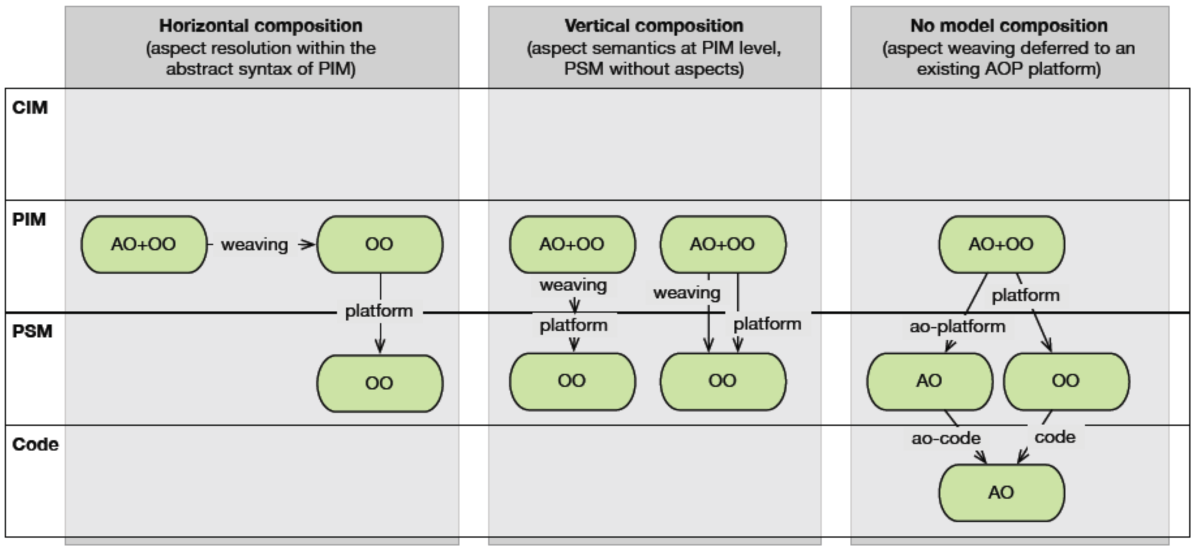


Figure 1: Different kinds of model composition

The different variations of aspect oriented composition are depicted in Figure 1. *Horizontal Composition*, which was chosen for the developed approach, processes the aspect weaving on the same abstraction level. The input and output models are (in the depicted case) models at the PIM level. The output model does conform to the metamodel of the base model, which leads to the fact, that the output model can be processed using common tools without support for aspects (e.g. using Objecteering [19] to produce Java code). Because the aspect weaving is done at the PIM level, no support of aspect-oriented concepts at PSM and Code level is required. The horizontal composition can be realized by adapting additional model-to-model transformations. The existing model compiler, which transforms the PIM level model to PSM level model or to code does not have to be adapted.

The *Vertical Composition* processes the aspect weaving during the transformations between different model levels (in Figure 1 PIM to PSM). The existing model compiler (respectively the model transformations) has to be adapted to support the aspect composition during the transformation between the abstraction levels. Therefore the extension of existing MDA processes by using the vertical composition cases needs more effort than the usage of vertical composition. Nevertheless, the support of aspect-oriented concepts at code level is not required in this case, too.

If *No Model Composition* is processed during the model transformation between different abstraction levels, the base and the aspect model are only transformed into the corresponding representation on the next level. No aspect weaving is done. The extension of this MDA process requires no adaptation of the transformations for the base models, but only the provision of additional transformations for the aspect models. This variant requires aspect oriented support at code level, because the aspect weaving is not processed during the MDA process, but rather has to be realized on code level (static/source code weaving, dynamic weaving, load time weaving, runtime weaving, etc.)

2.2.2 AO composition Variations

In the domain of the aspect oriented programming, different weaving approaches and strategies at different levels are developed. Aspect weaving can be processed at different points in time, e.g. static weaving, load time weaving, runtime weaving, etc. One of the challenges was, to investigate known weaving strategies and to decide, which strategy should be used for the developed aspect-oriented composition.

Since horizontal composition on PIM level was chosen, the weaving is done in a static way. The composition is done without runtime information.

There are two general possibilities to process the aspect weaving. First, the advice model can be inlined at the captured joinpoints (as shown in [15]). After the advice model is inlined, the aspect module does not exist explicitly in the woven model. Therefore the realisation of different instantiation strategies is hindered, since there is no explicit class, which could have different instantiation mechanisms. On the other hand, there are no additional infrastructural model artefacts.

The second approach is the encapsulation of the aspect and advice in a separate class (in the woven model), which can be instantiated as a singleton (see subsection 2.2.3). Different instantiation strategies can be realized during this aspect composition with less effort.

Additional strategies can be applied, for instance to increase the weaving performance. The approach in [20] extracts all potential joinpoint shadows to so called envelopes (Getter/Setter for fields and proxies for methods). The field and method accesses are replaced by calls to the corresponding envelopes. This “pre-processing” has the effect, that the search scope for the potential joinpoints is reduced, which is an optimisation for the pointcut matching phase. The number of points, where the aspect weaving takes place is reduced and the weaving process can be simplified.

Another problem in this domain is the handling of runtime properties of joinpoints, which are used in the pointcut declaration to define the set of selected joinpoints. During the pointcut resolving phase the dynamic properties of potential joinpoints can be approximated to decide, if a point in the execution is a joinpoint. This variant possibly requires a complex and time consuming static analysis, which can only approximate the runtime properties of potential joinpoints. The approach, which is used in AspectJ [21], checks the static properties during the pointcut resolving. The result is a set of *potential* joinpoints. During the aspect weaving phase, corresponding behaviour for checking the runtime properties is woven before the advice call. Only if the woven condition is true at runtime, the advice is executed.

This approach can also be applied for aspect composition on PIM level. The developed approach focuses on static approximation of runtime properties. For instance the type of the defined context exposure pointcut expressions is checked in a static way (see also subsection 4.3.3.2).

2.2.3 Instantiation Variations

Not only the weaving strategies but also the aspect instantiation strategy plays an essential role in the aspect composition. If the aspect stores context information, it is often useful to decide, if the called advice should have always access to the same context information (independently of the triggering object or of the object, on which the joinpoint is triggered).

In other words, the instantiation strategy decides, if an advice is always called on one aspect instance, or if each object has its own aspect instance (with separate context information) on which the advice is called, etc.

The realisation of a certain instantiation strategy partially depends on the chosen weaving strategy. If the aspect is encapsulated in a separate aspect class, the support of different instantiation strategies can be achieved by the provision of certain mechanisms for aspect class creation (singleton, hash table for associating several objects with corresponding instances of an aspect class, etc.).

If the content of an aspect (fields, advices, operations) is inlined at the corresponding joinpoints during the aspect composition, each object, which is adapted by the aspect, has its own context. The single aspect instance strategy cannot be realized, without the provision of additional model infrastructure.

Since the developed approach encapsulates the aspect in a separate class, the instantiation strategies “singleton” and “perThis” are supported.

2.3 Evaluation

In this section, we will use the software properties *understandability* and *maintainability* as evaluation criteria to compare the aspect-oriented modelling approach proposed in work package 3 with the traditional object-oriented modelling approach at the PIM level. For each evaluation criterion, we discuss some factors that affect that criterion and introduce a few metrics to measure those factors. Table 2 gives an overview of the different factors and metrics that we will use to evaluate the properties understandability and maintainability.

Property	Factor	Metric
Understandability	Size	Number of Actions
		Number of Model Elements
	Complexity	Cyclomatic Method Complexity
	Separation of Concerns	Concern Diffusion over Actions
Maintainability		Concern Diffusion over Modules
		Concern Diffusion over Operations
	Ease of Change	Number of Impacted Components
		Number of Impacted Members

Table 2: Overview of Evaluation criteria and the selected metrics

These evaluation criteria and the respective factors and metrics will be used in the next section together with two examples of crosscutting concerns to assess the proposed aspect-oriented modelling approach. The list of evaluation criteria and respective factors and metrics is by no way complete. Our evaluation is a first effort towards assessing the benefits of aspect-oriented modelling at the PIM level. A complete and extensive evaluation of aspect-oriented modelling is beyond the scope of WP3 and the VIDE project.

2.3.1 Understandability

This software property reflects how difficult understanding the application models is. It also includes understanding the way a crosscutting concern (i.e., the respective structure and behaviour) is modelled and its relation to the core business logic of the application. We selected *size*, *complexity*, and *separation of concerns* as some of the factors that affect understandability. Next, we present metrics for measuring these factors.

a) Size Metrics:

- ***Number of Actions (NoA):*** the total number of UML actions in a method body. For a class this metric is the sum of the number of actions of its methods and constructors. Inherited methods are not included.
- ***Number of Model Elements (NoME):*** In addition to measuring the number of actions this metrics includes also the number of control flows and object flows in the behaviour model of a method. For a class, this metric can be calculated as the sum of the NoME values of its methods and constructors.

b) Complexity Metrics:

Cyclomatic Method Complexity (CC): this metric was introduced by McCabe to measure the flow complexity of a method [1]. It is the number of linearly independent paths and consequently gives information on the minimum number of paths that have to be tested. It basically counts the number of places in the method body where the flow changes from a linear flow (e.g., in if then statement, loops, etc). To measure this complexity, we will proceed as described in [2], which proposes a simple way to count this metric: one starts with a count of one for the method and adds one for each of the flow-related elements that are found in the method body such as selection (such as *if then else* and *switch*), loops (such as *for* and *while*), and logical operators (such as the operators *and* and *or*).

c) Separation of Concerns Metrics:

In [4], Sant' Anna et al. introduced three metrics for measuring the separation of concerns. One of these metrics is called *Concern Diffusion over Lines of Code* (CDLOC) and it is especially relevant in the context of understandability. The two other metrics are more important with regard to maintainability.

Concern Diffusion over Actions (CDA) is an adapted metric for VIDE that is based on the separation of concerns metric *Concern Diffusion over Lines of Code* (CDLoC) [4]. It counts the number of transition points for each concern through the actions of the behaviour models. Transition points are points in the behaviour of a method or a constructor where there is a "concern switch" for instance from business logic to security.

2.3.2 Maintainability

This software property reflects how easy/hard and time-consuming the process of maintaining the software is. For the software to be maintainable it should be easy to understand, to enhance, to extend or to correct. Several factors have an impact on maintainability such as ease of change, separation of concerns, complexity and size. Maintainability and understandability are also related to a large extent as it is quite hard to maintain an application that is not understandable. As a result, the complexity and size metrics that were introduced in the last subsection can be used also for measuring the maintainability of the application models.

In the following, we will focus mainly on maintainability with respect to crosscutting concerns. That is, we will measure how difficult it is to perform changes to the behaviour models corresponding to crosscutting concerns. Thereby, we will concentrate on two maintainability factors that were defined in [5]:

a) Ease of Change Metrics:

These metrics measure the difficulty level in changing the modelling elements that belong to a crosscutting concern, e.g., to customize or extend the application. The following two metrics give an idea on the scope of the change. In addition to that, it is also important to consider the time aspect, i.e., how long does it take to perform a certain change.

- ***Number of Impacted Components (NIC)***: this metric counts the number of classes and aspects that are affected by a certain change [5].
- ***Number of Impacted Members (NIM):*** this metric counts the number of operations and attributes that are affected by a certain change [5].

b) Separation of Concerns Metrics:

In addition to *Concern Diffusion over Lines of Code* (CDLOC), Sant' Anna et al. introduced two other metrics for measuring the separation of concerns. These metrics are important with respect to understandability and maintainability.

- ***Concern Diffusion over Modules (CDM):*** This metric counts the number of classes and aspects that contribute to the implementation of a concern as well as the number of other classes and aspects that access them [4]. In this document, we focus only on the number of classes and aspects that contribute to the implementation of a concern.

- ***Concern Diffusion over Operations (CDO)*** counts the number of methods and advices that contribute to the implementation of a concern and the number of other methods and advices that access them [4]. In this document, we focus only on the number of methods and advices that contribute to the implementation of a concern.

3 Business Scenario

This section starts by a short review of opportunity management, which is part of the Customer Relationship Management (CRM) business application that was presented in D3.1. After that, the crosscutting concerns consistency checks and partner determination will be modelled once with and once without aspects and evaluation data will be collected using the evaluation factors and metrics that were defined in Section 2.3. A discussion of gathered evaluation data will then follow.

3.1 Review of the Opportunity Scenario

Customer Relationship Management [6] is a management concept, which intends to systematize and improve the relationships between companies and their customers. It is a customer-oriented corporate strategy that utilises modern information and communication technologies to establish long-term, profitable customer relationships by providing a central tool that integrates marketing, sales and service instruments [6]. SAP offers several CRM products such as SAP CRM [8,9], which covers the three fundamental CRM processes marketing, sales, and service.

Figure 2 shows some typical pre-sales and sales processes in an enterprise that sells one or more products. These processes involve different steps such as opportunity management, quotations to customers, sales orders and invoice processing. This figure shows also the different user roles that are involved in each process step.

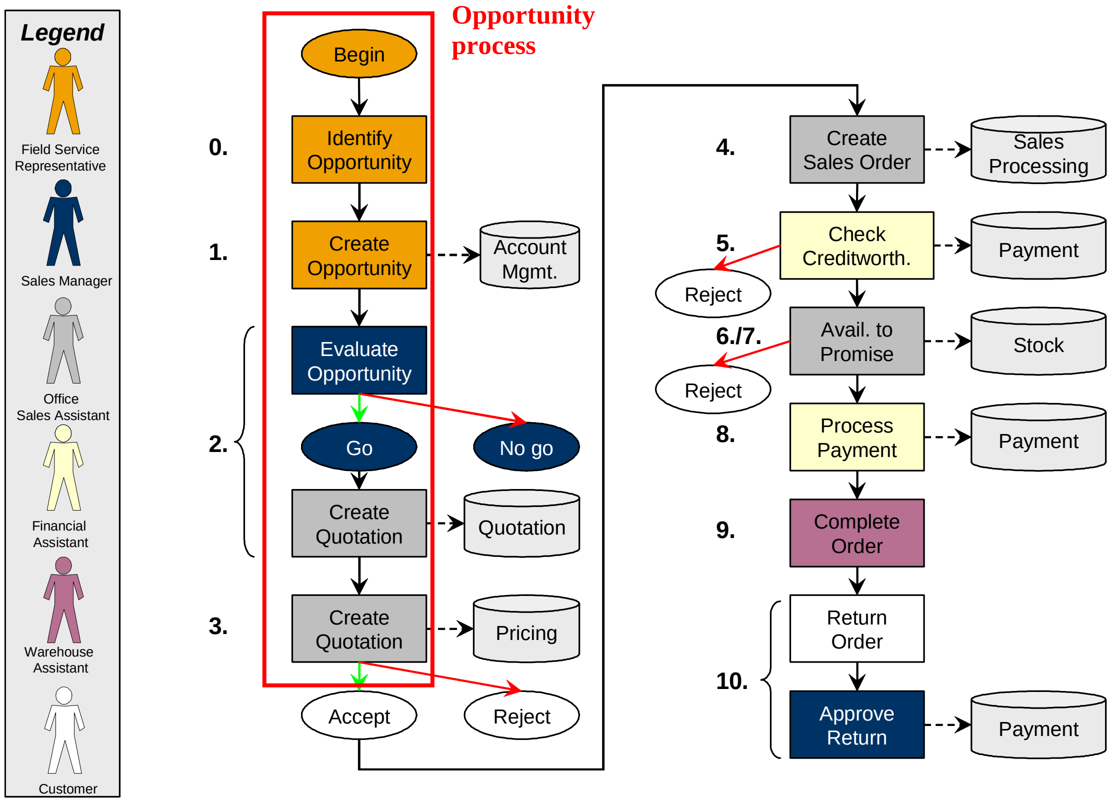


Figure 2: Sales Scenario

3.1.1 Opportunity Management

Opportunity management is a pre-sales process that provides a structured approach to turning an initial recognition of a selling opportunity into a sales contract. In that process, which is shown in Figure 2, the SAP CRM software guides the sales representative through a process and generates next steps and activity suggestions on the basis of best-practice sales strategies.

The SAP CRM business application is implemented as an object-oriented application. Figure 3 shows an extract of a class diagram with the main business objects that are involved in opportunity management. More details on some of these objects can be found in D3.1.

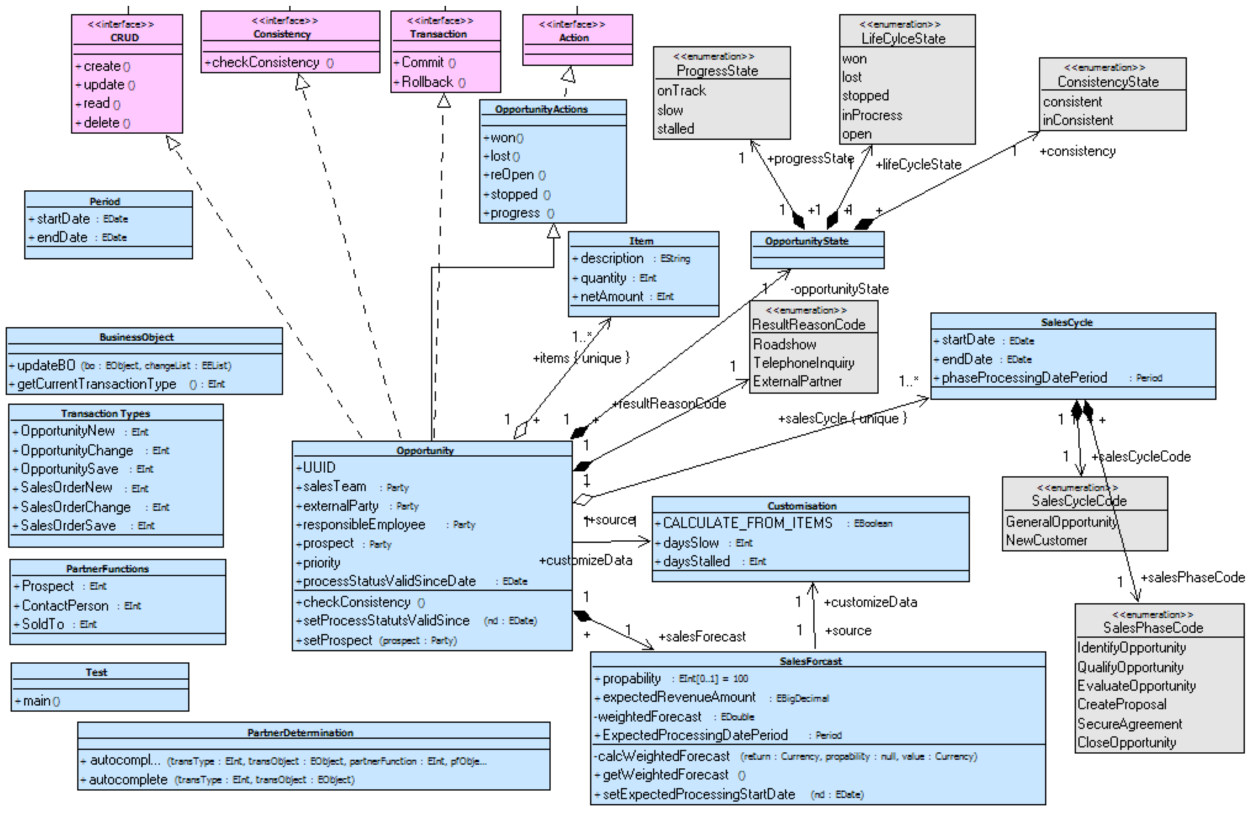


Figure 3: Main Classes in Opportunity Management

3.2 Modelling Crosscutting Concerns in the Opportunity Scenario

In D3.1, *consistency checks* and *partner determination* were introduced as examples of crosscutting concerns in opportunity management. In this section, we model each of these concerns once without aspects and once with aspects and thereby use the evaluation factors and metrics defined in Section 2.3 to evaluate and compare both approaches with respect to understandability and maintainability.

3.2.1 Consistency checks

Several consistency checks are performed when the state of the opportunity business object or some of the associated objects changes. The code that enforces consistency checks cuts across different classes.

In D3.1, consistency constraints were classified into simple constraints and complex constraints based on the degree of crosscutting. The enforcement of the complex consistency constraints involves more than one business object class, e.g., the constraint C3 is a complex constraint that should be fulfilled by each Opportunity object to be in a consistent state.

```
(C3):  Opportunity.processStatusValidSinceDate <
       SalesForecast.expectedProcessingDatePeriod.StartDate
```

Appropriate logic is needed to check consistency constraints such as C3 and hinder their violation. This logic should be triggered when the fields corresponding to the constraint are modified and/or when the respective setter methods are called. Consequently, it is scattered across several classes. For instance, to enforce the constraint C3, appropriate logic is required in the method *setProcessStatusValidSinceDate* of the class *Opportunity* to check that the date is smaller than *expectedProcessingDatePeriod.StartDate* in the associated *SalesForecast* object as shown below in Java.

```
 //defined in class Opportunity
 public void setProcessStatusValidSince(Date nd)
{
      if(this.salesForecast.expectedProcessingDatePeriod.startDate > nd)
      this.processStateValidSinceDate = nd;
}
```

Similar logic is also needed in the method *setExpectedProcessingDatePeriod* to verify that the *StartDate* of the new period is smaller than the value of the attribute *processStatusValidSince* of the associated *Opportunity* object as shown below in Java.

```
//defined in class SalesForecast
public void setExpectedProcessingStartDate (Date nd)
{
     if(nd > this.opportunity.processStatusValideSinceDate)
     this.expectedProcessingStartDatePeriod.startDate = nd;
}
```

3.2.1.1 Modelling consistency checks without aspects

In the following, we model the constraint C3 using UML actions but without using aspects at the PIM level.

A) The models

Figures 4 and 5 show the behaviour models that correspond to the method bodies of *setExpectedProcessingStartDate*, which is defined in the class *SalesForecast* and *setProcessStatusValidSince*, which is defined in the class *Opportunity* respectively. These models were drawn using the tool TopCased [10].

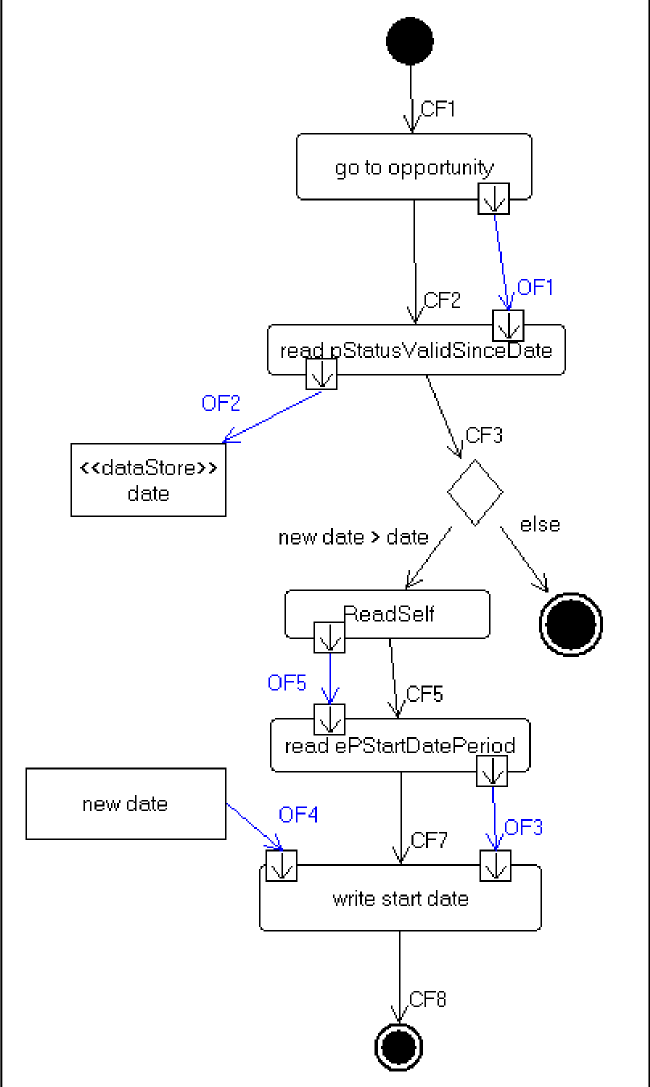


Figure 4: Method setExpectedProcessingStartDate

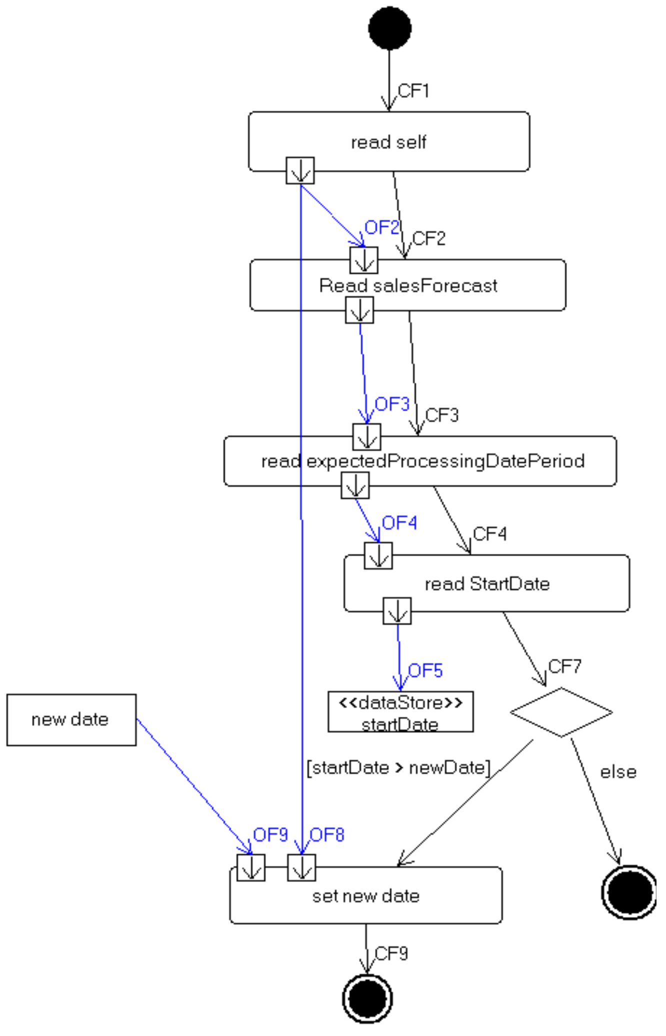


Figure 5: Method setProcessStatusValidSince

B) Measurement Data

Next, we use the metrics presented in Section 2.3 to collect quantitative data on the understandability and maintainability of this model.

- Size
    - Number of Actions:
        - 5 in method *setExpectedProcessingStartDate*
        - 5 in method *setProcessStatusValidSince*
    - Number of Model Elements:

        - **18** in method *setExpectedProcessingStartDate* (8 control flows, 5 object flows, 5 actions)
        - **19** in method *setProcessStatusValidSince* (8 control flow, 6 object flows, 5 actions)

- Complexity
    - Cyclomatic Complexity
        - **2** for method *setExpectedProcessingStartDate*
        - **2** for method *setProcessStatusValidSince*
- Separation of Concerns
    - Values of Concern Diffusion over Actions for the concern consistency checks
        - **2** in the method *setExpectedProcessingStartDate*
        - **2** in method *setProcessStatusValidSince*
    - Concern Diffusion over Operations value of **2** for the specific consistency check C3 (much more for consistency checks as one concern in the opportunity application)
    - Concern Diffusion over Modules value of **2** for the specific consistency check C3, which spans the classes Opportunity and SalesForecast (a much higher CDC value if consistency checks in general are considered as one concern in the opportunity application)
- Ease of Change

We assume that the consistency check C3 and its implementation have to be changed. For example, the constraint may be relaxed to not longer require *processStatusValidSinceDate* to be strictly less than the start date of *expectedProcessingDatePeriod* as said before but only less or equal. The values for the metrics related to this change are as follows.

- Number of Impacted Components: **2** (namely the classes Opportunity and SalesForecast)
- Number of Impacted Members: **2** (namely the two setter methods that are covered by this constraint)

3.2.1.2 Modelling consistency checks with aspects

Next, we model the constraint C3 using an aspect. This aspect consists of a pointcut that selects two join points (i.e., the execution of the two setter methods) and two advices that define the logic for enforcing the consistency constraint.

A) The Models

Figure 6 shows an aspect that modularizes the consistency check C3. This figure also shows the pointcut of this aspect, which selects two join points in the base. More precisely, this pointcut selects all write accesses (kind = set) to properties that have the type Date (pce1.type.namePattern = “Date”) and which belong to objects of type “Opportunity” (pce1.declaringType.namePattern = “Opportunity”) as well as write accesses to properties that have the type Date and which belong to objects of the type “SaleForecast”.

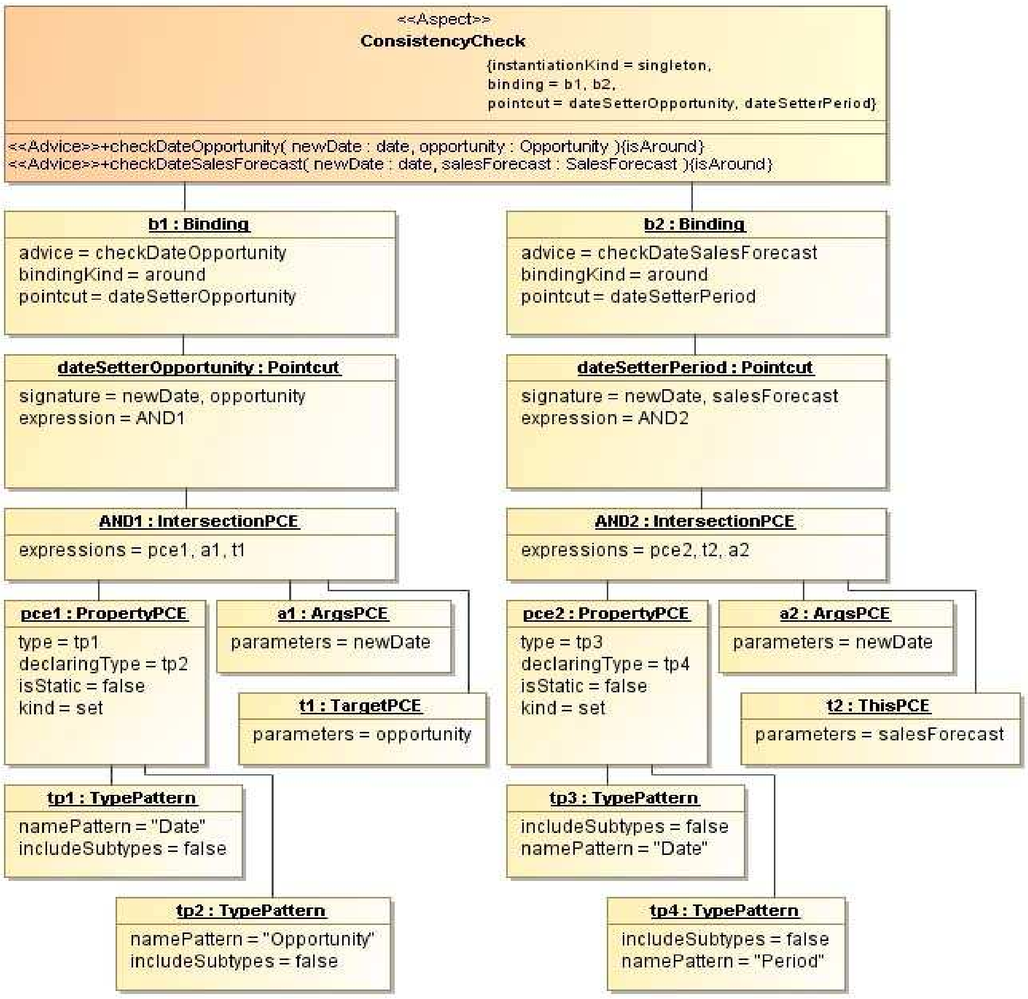


Figure 6. Aspect for constraint C3

Figures 7 and 8 show the two advices of this aspects. These advices are bound “around” the selected join points in order to control the original behaviour (that sets the attributes related to C3) and execute it only when the constraint C3 is fulfilled.

Figures 9 and 10 show the behaviour models of *setExpectedProcessingStartDate* and *setProcessStatusValidSince*. These models do not contain any logic for enforcing consistency checks because this logic is now externalized in a separate aspect. That is these two methods just set the appropriate attribute to the new date that is passed as a parameter.

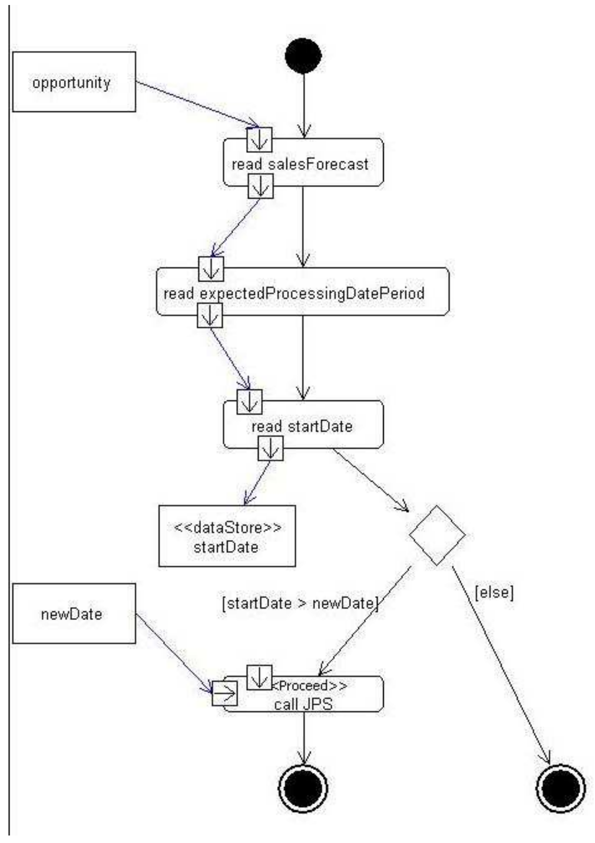


Figure 7: Advice for Enforcing C3 in Opportunity

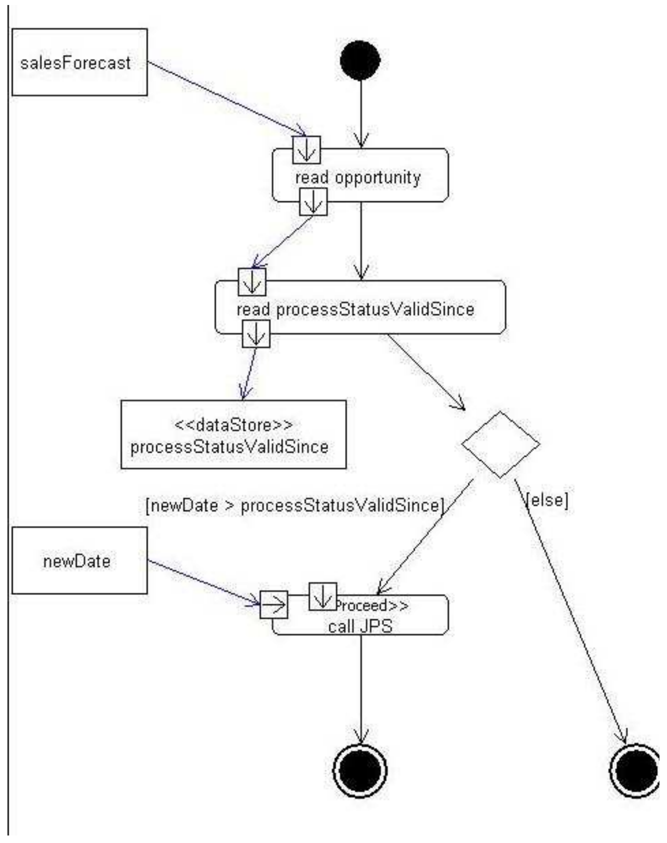

salesForecast
read opportunity
read processStatusValidSince
<<dataStore>> processStatusValidSince
[newDate > processStatusValidSince]
[else]
newDate
<<Proceed>> call JPS

Figure 8: Advice for Enforcing C3 in SalesForecast

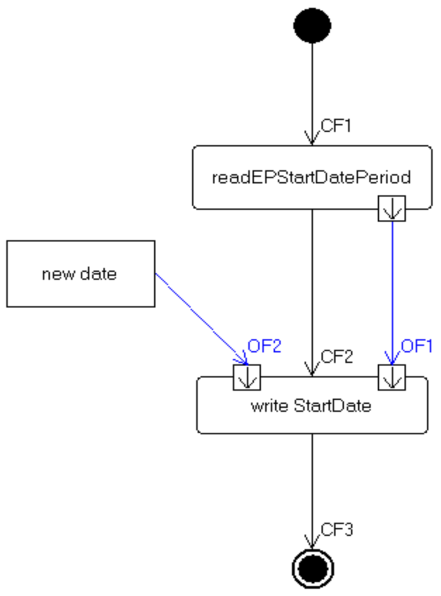


Figure 9: Method setExpectedProcessingStartDate

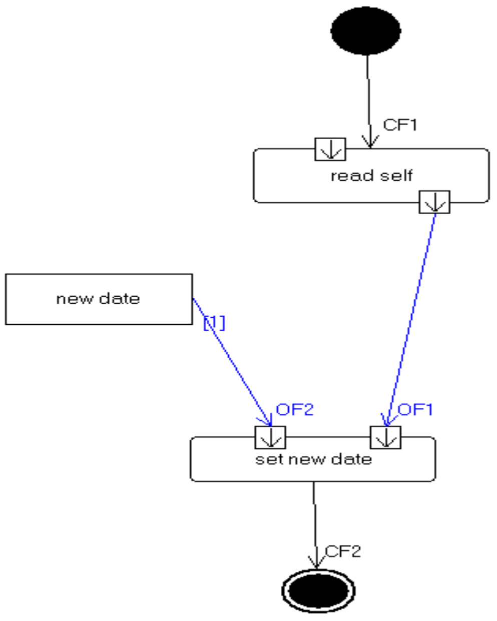


Figure 10: Method setProcessStatusValidSince

B) Measurement Data

Next, we provide quantitative values to measure the understandability and maintainability of the aspect-oriented design using the metrics presented in Section 2.3.

- Size
  - Number of Actions:
    - **2** in method setExpectedProcessingStartDate
    - **2** in method setProcessStatusValidSince
    - **4** in advice for enforcing C3 on Opportunity objects
    - **3** in advice for enforcing C3 on SalesForecast objects
  - Number of Model Elements:
    - **7** in method *setExpectedProcessingStartDate* (3 control flows, 2 object flows, 2 actions)
    - **6** in method *setProcessStatusValidSince* (2 control flow, 2 object flows, 2 actions)
    - **16** in advice for enforcing C3 on Opportunity objects (7 control flow, 5 object flows, 4 actions)
    - **13** in advice for enforcing C3 on SalesForecast objects (6 control flow, 4 object flows, 3 actions)
- Complexity
  - Cyclomatic Complexity
    - **1** for method *setExpectedProcessingStartDate*
    - **1** for method *setProcessStatusValidSince*
    - **2** in advice for enforcing C3 on Opportunity objects
    - **2** in advice for enforcing C3 on SalesForecast objects

- Separation of Concerns
  - Concern Diffusion over Actions for the concern consistency checking
    - **0** for the method *setExpectedProcessingStartDate*
    - **0** for method *setProcessStatusValidSince*
  - Concern Diffusion over Operations value of **2** for the specific consistency check C3, which is now implemented using two advices that are modularized in one aspect
  - Concern Diffusion over Modules value of **1** for the specific consistency checks C3, which is modularized now in one aspect.

- <u>Ease of Change</u>

Changing the consistency check C3 and the respective implementation then

- Number of Impacted Components: **1** (namely the aspect)
- Number of Impacted Members: **2** (namely the two advice)

3.2.2 Partner determination

Partner determination [11,12] is part of the partner processing function of many SAP business applications. It refers to the system ability to automatically find and enter partner information such as addresses in certain transactions and documents. That is, the user enters manually one or more partners 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 such as the last document or the parent document, etc.

Figure 11 shows an example that illustrates how partner determination works. The user creates an opportunity and enters the name of the sales prospect whereas 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).

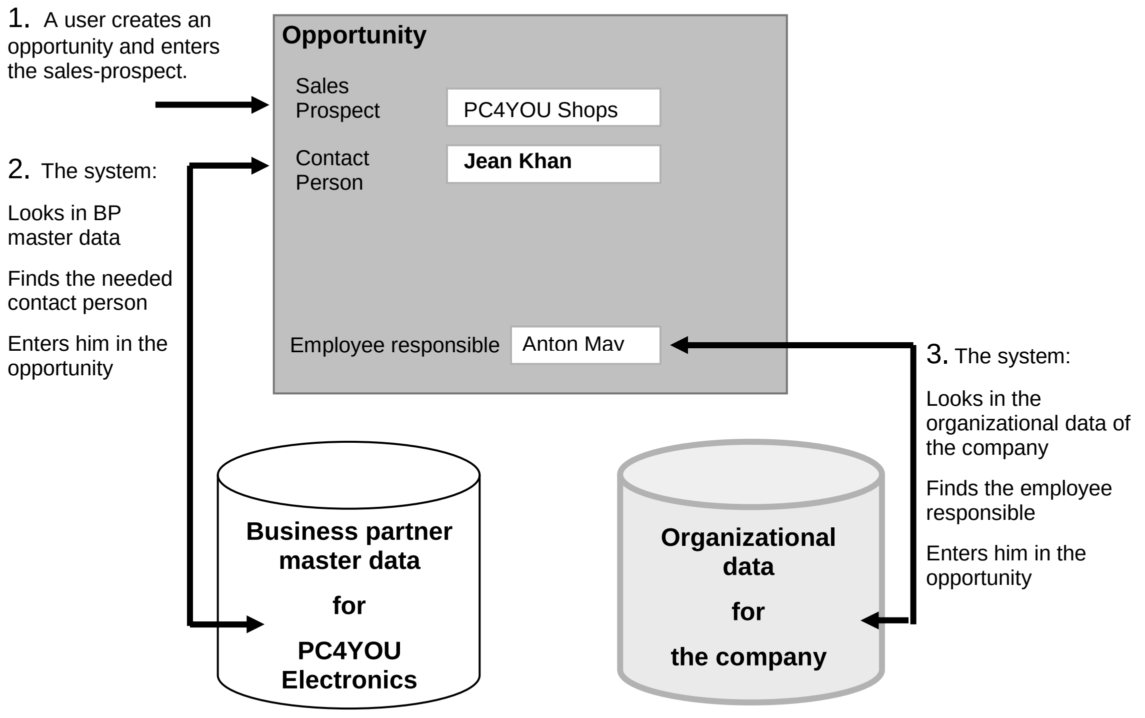

1. A user creates an opportunity and enters the sales-prospect.
Opportunity
Sales Prospect
PC4YOU Shops
Contact Person
Jean Khan
2. The system:
Looks in BP master data
Finds the needed contact person
Enters him in the opportunity
Employee responsible
Anton Mav
3. The system:
Looks in the organizational 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 11: Partner Determination in Opportunity Management

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 can configure the way partner determination is run for a certain transaction by defining *partner determination procedures*. The latter define which *partner functions* are mandatory or optional for a given transaction. For each *partner function* (e.g., contact person, sold-to-party, ship-to-party, etc.) the users can specify the sources of information that should be used to determine the partner function values and in what order these sources are searched (so-called *access sequences*). They can also configure when partner determination is performed, e.g., when data is entered by the user or when it is saved.

Partner determination is a crosscutting concern as the respective code is scattered across several classes of the user interface and the business objects of the CRM application. Partner determination may be triggered in the UI classes, e.g., when the user enters the sales prospect for a new opportunity and can be also triggered in the business object, e.g., when an opportunity object is saved (i.e., the update method of the CRUD interface is called). Partner determination functionality logic is scattered over other business object classes in the CRM application such as *Opportunity* and *SalesOrder*.

When a new opportunity is created and the user sets the prospect, partner determination will automatically add the contact person at the partner and the responsible employee within the partner organization. There is some logic in the method *setProspect*, which triggers automatic partner determination. Later, if these partner functions are changed by the user no automatic update will be executed. However, there are some other partner determination procedures that may be triggered automatically when an opportunity is updated (the method *update* of the CRUD interface).

When a new SalesOrder is created and the user enters the Sold-to-party (method *setSoldTo*), partner determination is triggered and some partner functions are completed automatically such as Ship-to-party, Bill-to-party, Payer, Contact Person, and Responsible Employee. Similarly to the update method of the opportunity class, some partner determination logic is executed when a sales order is updated (method *update*).

3.2.2.1 Modelling Partner determination without aspects

In the following, we model the partner determination logic in the methods *setProspect*, *setSoldTo*, and *update* without using aspects.

A) The Models

Figures 12 and 13 show the behaviour models of the methods *setProspect* and *update* in the class Opportunity. Below is the equivalent behaviour of these two methods in Java.

```
public void setProspect (Party pros) //defined in the class Opportunity
{
    this.prospect = pros;
    //run partner determination procedure for business function sales prospect
    if(getCurrentTransactionType()==TransactionType.OpportunityNew)
         PartnerDetermination.autocomplete(TransactionTypes.OpportunityNew,this,
         PartnerFunctions.Prospect, prospect);
}
public void update(List changelist) //defined in the class Opportunity
{
   //call update on the parent
  BusinessObject.updateBO(this, changelist);

  //run partner determination procedure
  PartnerDetermination.autocomplete(TransactionTypes.OpportunityChange, this);
}
```

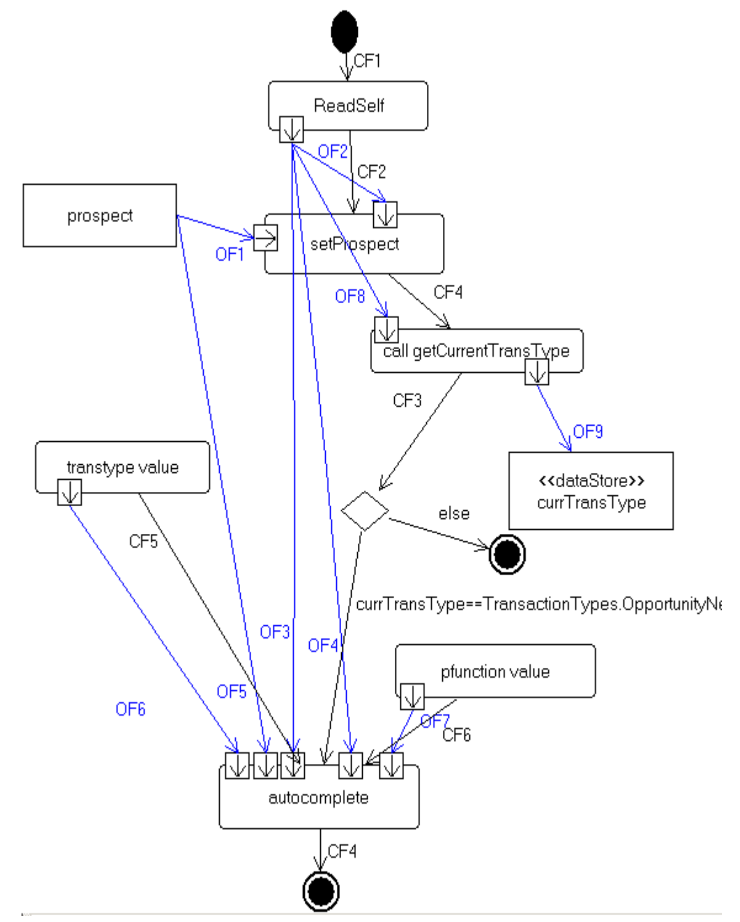


Figure 12: Method setProspect

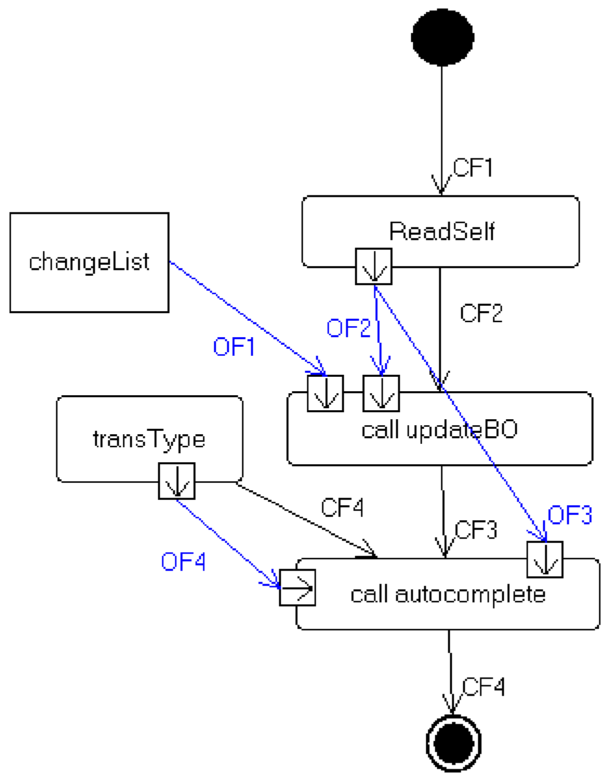


Figure 13: Method update

Figures 14 and 15 show the behaviour models of the methods *setSoldTo* and *update* in the class SalesOrder. Below is the equivalent behaviour of these two methods in Java.

```
//defined in the class SalesOrder
public void setSoldTo(Party soldTo)
{
     this.soldTo = soldTo;
    //run partner determination procedure
     if(getCurrentTransactionType()==TransactionType.SalesOrderNew)
     {
       PartnerDetermination.autocomplete(TransactionTypes.SalesOrderNew,this,
      PartnerFunctions.SoldTo, soldTo);
    }
}

//defined in the class SalesOrder
public void update(List changelist)
{
    //call update on the parent
   BusinessObject.updateBO(this, changelist);

   //run partner determination procedure
   PartnerDetermination.autocomplete(TransactionTypes.SalesOrderChange, this);
}
```

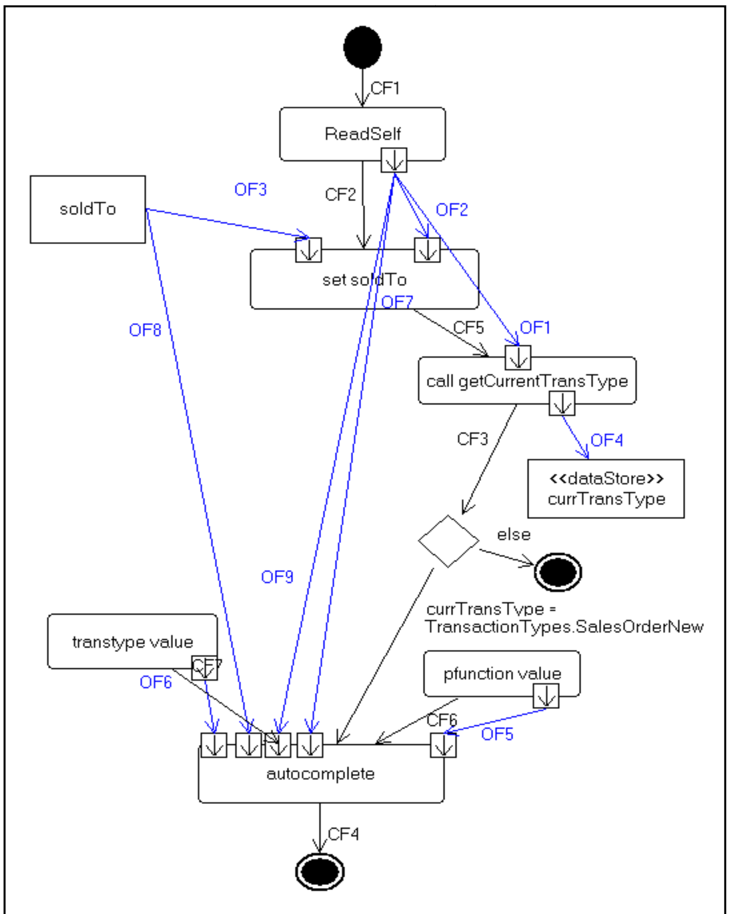


Figure 14: Method setSoldTo

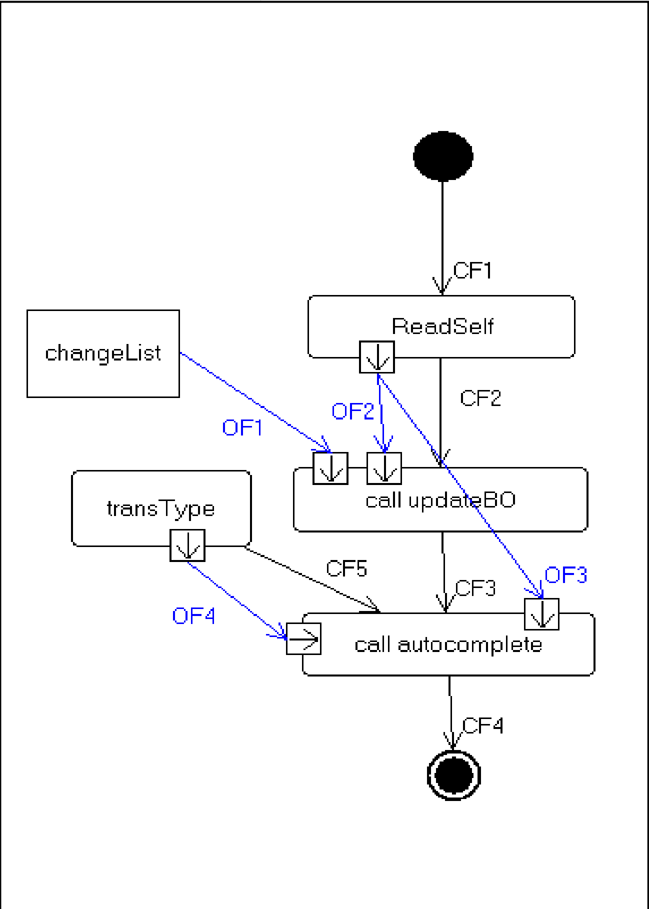


Figure 15: Method update

B) Measurement Data

Next, we provide some quantitative data on the understandability and maintainability of this design using the metrics that were presented in Section 2.3.

- Size
    - Number of Actions:
        - **6** in method setProspect (class Opportunity)
        - **4** in method update (class Opportunity)
        - **6** in method setSoldTo (class SalesOrder)
        - **4** in method update (class SalesOrder)

    - Number of Model Elements:
        - **24** in method setProspect of the class Opportunity (9 control flows, 9 object flows, 6 actions)
        - **11** in method update of the class Opportunity (5 control flow, 4 object flows, 4 actions)

        - **24** in method setSoldTo of the class SalesOrder (9 control flows, 9 object flows, 6 actions)
        - **13** in method update of the class SalesOrder (5 control flow, 4 object flows, 4 actions)

- <u>Complexity</u>
    - Cyclomatic Complexity
        - **2** for method setProspect
        - **1** for method update of class Opportunity
        - **2** for method setSoldTo
        - **1** for method update of class SalesOrder

- <u>Separation of Concerns</u>
    - CDA for the concern partner determination
        - **2** in the method setProspect
        - **2** in the method update of class Opportunity
        - **2** for method setSoldTo
        - **2** for method update of class SalesOrder
    - CDO value of **4** at least for the considered extract of the opportunity scenario models as behaviour relating to partner determination is found in four methods.
    - CDM value of **2** at least as partner determination behaviour is scattered over the classes Opportunity and SalesOrder.

- <u>Ease of Change</u>

  We assume that the partner determination logic should be extended in some way to e.g., fire an event after the automatic completion. The impact of this change is as follows:
    - Number of Impacted Components: **2** (the classes Opportunity and SalesOrder)
    - Number of Impacted Members: **4** (as 4 operations are involved)

3.2.2.2 Modelling Partner determination with aspects

Next, we model partner determination using an aspect.

A) The Models

Figure 16 shows the aspect model for partner determination. This aspect defines two bindings: The binding *b1* connects the pointcut updateBO, which selects the calls to the operation *updateBO*, to the advice *updateChange* that is shown in Figure 17; the binding *b2* connects the pointcut partySetter, which selects calls to the operations *setSoldTo* (class SalesOrder) and *setProspect* (class Opportunity), to the advice *udpateNew* that is shown in Figure 18. Both advices are executed after the selected join points.

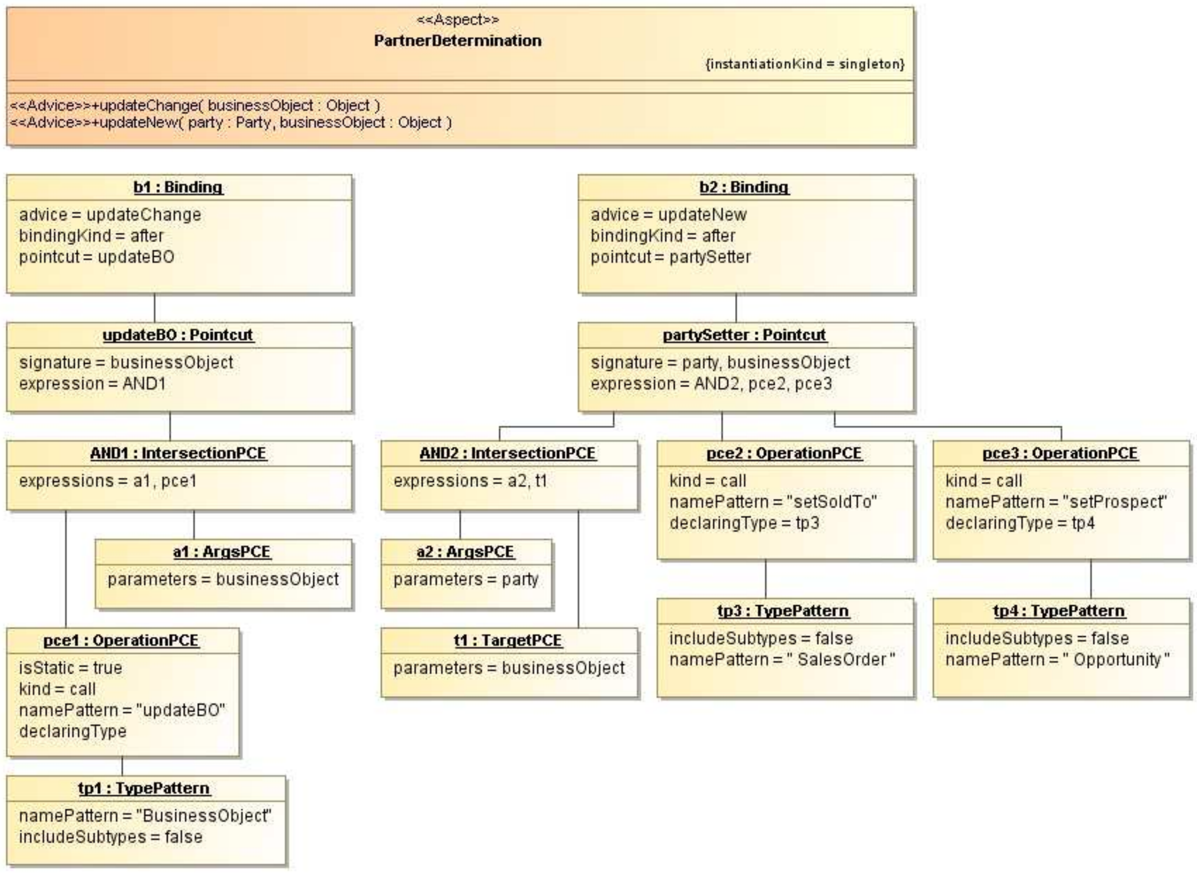


Figure 16: Partner Determination Aspect

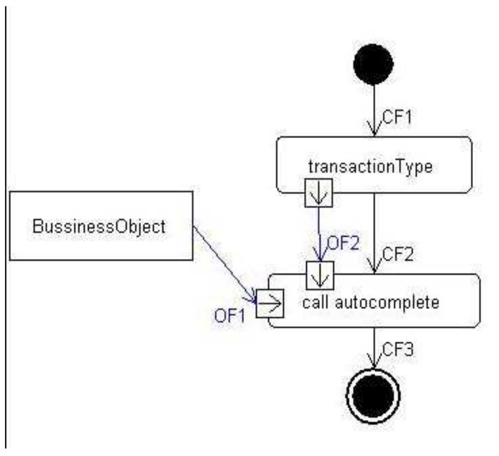


Figure 17: Advice updateChange

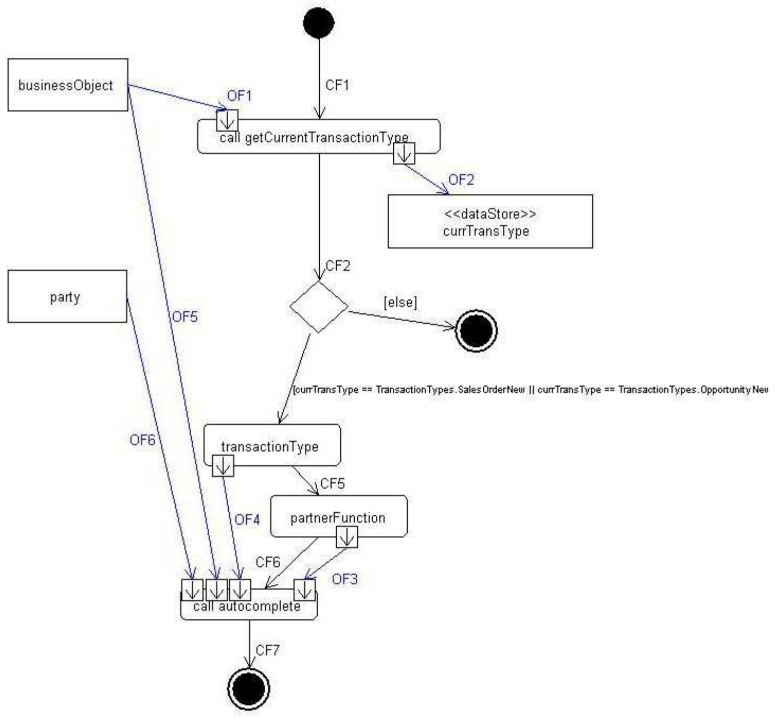


Figure 18: Advice updateNew

Figures 19 and 20 show the behaviour models of the methods *setProspect,* and *update* of the class Opportunity. Figures 21 and 22 show the behaviour models of the methods of *setSoldTo* and *update* of the class *SalesOrder*. These four models do not contain any logic for partner determination as this logic is now externalized in the aspect.

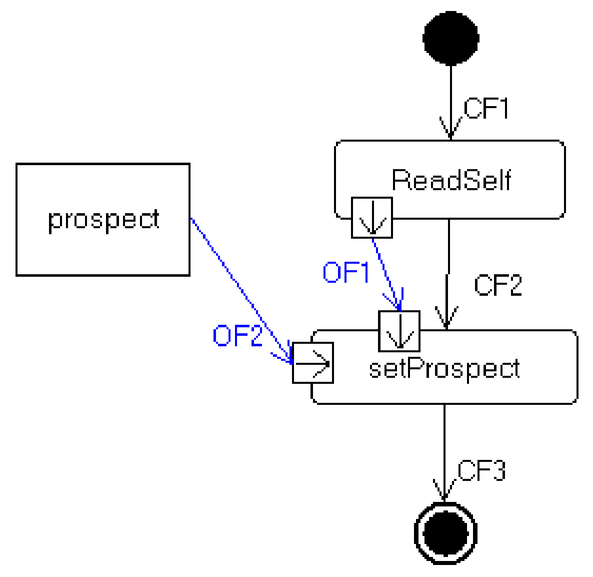


Figure 19: Method setProspect

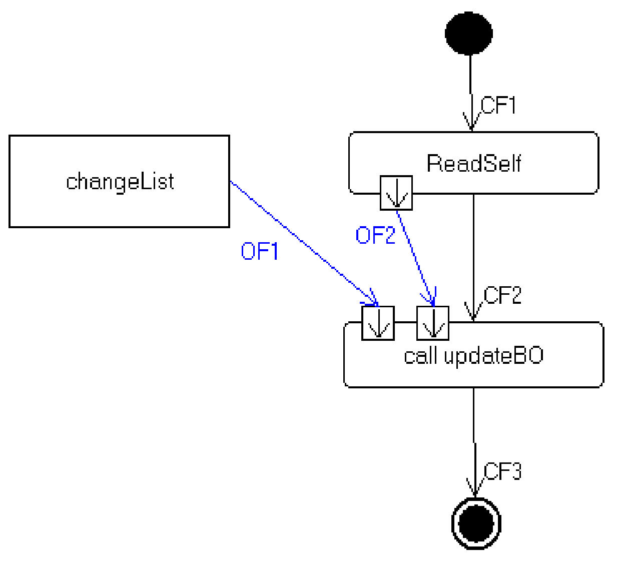


Figure 20: Method update

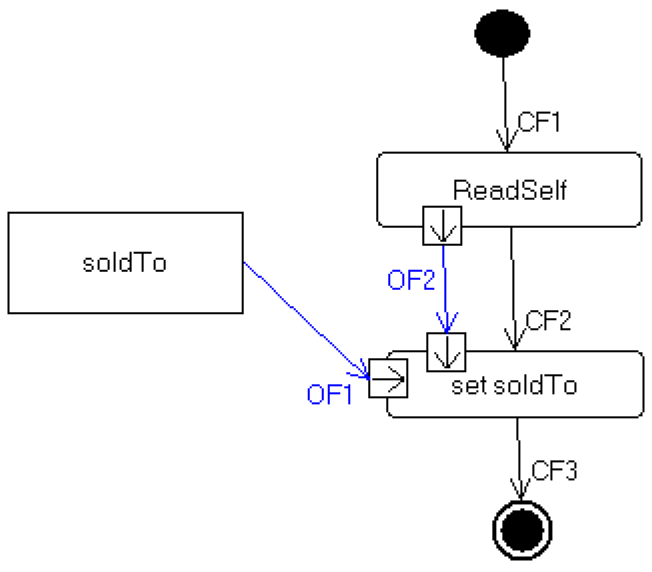


Figure 21: Method setSoldTo

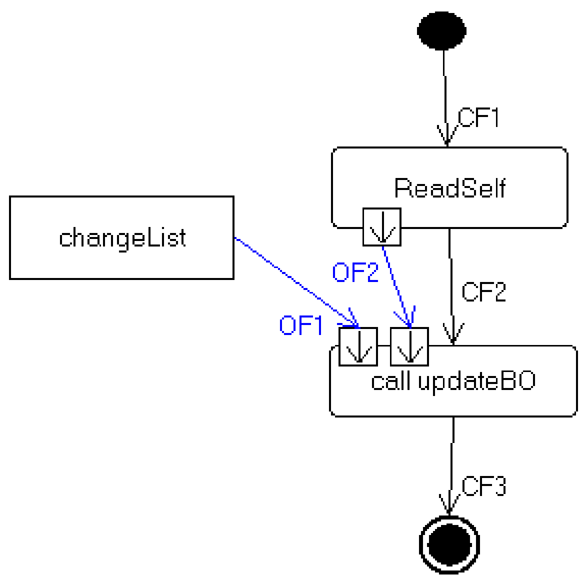


Figure 22: Method update

B) Measurement Data

In the following, we measure the values of the different metrics that were presented earlier to evaluate the understandability and maintainability of this aspect-oriented design.

- Size
    - Number of Actions:
        - **2** in method *setProspect* (class Opportunity)
        - **2** in method *update* (class Opportunity)
        - **2** in method *setSoldTo* (class SalesOrder)
        - **2** in method *update* (class SalesOrder)
        - **2** in advice *updateChange*
        - **4** in advice *updateNew*
    - Number of Model Elements:
        - **7** in method *setProspect* of the class Opportunity (3 control flows, 2 object flows, 2 actions)
        - **7** in method *update* of the class Opportunity (3 control flows, 2 object flows, 2 actions)
        - **7** in method *setSoldTo* of the class SalesOrder (3 control flows, 2 object flows, 2 actions)
        - **7** in method *update* of the class SalesOrder (3 control flows, 2 object flows, 2 actions)
        - **7** for advice *updateChange* (3 control flows, 2 object flows, 2 actions)
        - **17** for advice *updateNew* (7 control flows, 6 object flows, 4 actions)
- Complexity
    - Cyclomatic Complexity
        - **1** for method *setProspect*
        - **1** for method *update* of class Opportunity
        - **1** for method *setSoldTo*
        - **1** for method *update* of class SalesOrder
        - **1** for advice *updateChange*
        - **2** for advice *updateNew*
- Separation of Concerns
    - Concern Diffusion over Actions for the concern partner determination
        - **0** in the method *setProspect*
        - **0** in the method *update* of class Opportunity
        - **0** for method *setSoldTo*

        - **0** for method update of class *SalesOrder*
    - Concern Diffusion over Operations (CDO) value of **2** for this scenario as the partner determination logic is implemented in two advice, which are part of one aspect.
    - Concern Diffusion over Modules value of **1** as the partner determination behaviour is now modularized in one aspect.

- <u>Ease of Change</u>

We assume that the partner determination logic should be extended in some way to e.g., fire an event after the automatic completion. The impact of this change is as follows:

- Number of Impacted Components: **1** (the partner determination aspect)
- Number of Impacted Members: **2** (the advice *updateChange* and *updateNew*)

3.3 Evaluation

In the previous sections we modelled the crosscutting concerns consistency checks and partner determination respectively once with object-oriented PIM models and once with aspect-oriented models as proposed in VIDE. We will next discuss the measured evaluation data to compare both designs with respect to understandability and maintainability.

3.3.1 Understandability

We observe that the size of the methods *setExpectedProcessingStartDate* and *setProcessStatusValidSince* has been reduced drastically when the logic for enforcing the constraint C3 is externalized into an aspect. In fact, the number of actions decreased from 5 to 2 actions as well as the number of model elements which went down from 18 and 19 to 7 and 6 respectively. Moreover, the cyclomatic complexity of these two methods decreased as the logic for enforcing C3 is no longer part of them. The value of concern diffusion over actions is also reduced from 2 to 0 when C3 is modularized as an aspect because there is no concern switch in the two method bodies. All these metrics show that the understandability of the models is improved when aspects are used.

Similar observations are made in the case of partner determination. The size of the methods update, setProspect, and setSoldTo went down when the aspect is used to modularize partner determination. For instance, the number of model elements in the method bodies of *setSoldTo* and *setProspect* went down from 23 to 7 for each of them. The advice *updateNew*, which is called after the party setting in both methods, has a number of model elements value of 17. Moreover, the values of cyclomatic complexity for these methods also decreased when partner determination is modelled as an aspect. In addition, the value concern diffusion over actions went down from 2 to 0.

On the other hand, the usage of aspects adds additional complexity as the user has to understand the pointcut and what joint points in the behaviour models it matches. However, tools can be developed for that purpose. Such tools already exist for AspectJ [13,14].

3.3.2 Maintainability

The evaluation data shows that the maintainability of the application models is improved, which is as expected because of the better separation of concerns. For instance, the concern diffusion over modules went down from 2 to 1 for the specific consistency check C3. If other consistency checks are also considered the CDM value will even go down from higher values (i.e., the number of classes where logic for enforcing consistency checks is contained) to 1. Similarly, the concern diffusion over operation (CDO) for partner determination went down from CDM value 4 to 2 whereas the value of concern diffusion over modules went down from 2 to 1 when an aspect is used.

As a result of the improved separation of concerns the cost in effort and time for finding the model elements that implement the logic belonging to consistency checks is reduced. This is also reflected by the metric number of impact modules, which went down to 1 in both the consistency check and partner determination examples as only the aspect has to be changed. The number of impacted members also went down from 4 to 2 in the case of partner determination

3.4 Summary

In this section, we introduced opportunity management as a business scenario from SAP CRM applications. We presented two examples of crosscutting concerns there: consistency checks and partner determination. After that, we modelled each of these two concerns once without aspects and once with aspects and compared the two design options with respect to understandability and maintainability.

4 Specification of Aspect Oriented Composition in VIDE

This section describes the proposed integration of aspect-oriented concepts into MDD in the VIDE context. The integration can be split into several parts, which are outlined in the next section. The specification, described in this section, is based on the results and experiences collected during the research work, the Demonstrator development and the discussion with project partners.

4.1 Overview

To support aspect-oriented concepts in VIDE, contributions in different areas are necessary. Figure 23 depicts the relevant VIDE components, which have to be considered during the integration.

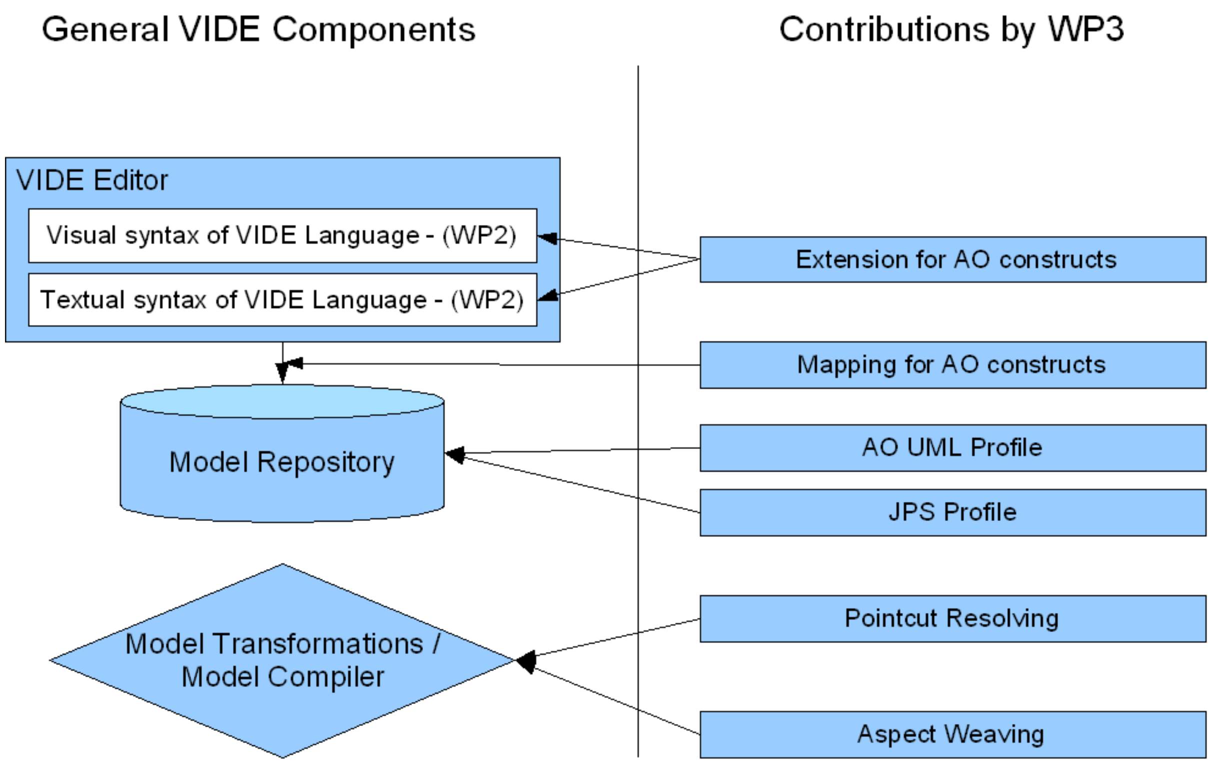

General VIDE Components
Contributions by WP3
VIDE Editor
Visual syntax of VIDE Language - (WP2)
Textual syntax of VIDE Language - (WP2)
Model Repository
Model Transformations / Model Compiler
Extension for AO constructs
Mapping for AO constructs
AO UML Profile
JPS Profile
Pointcut Resolving
Aspect Weaving

Figure 23: Contributions by WP3

The main part, we focus on, is the VIDE PIM language, which was specified in Work package 2 (VIDE/UML Metamodel). The corresponding model is stored in the EMF model repository. To integrate the aspect oriented composition in the VIDE PIM language, it is necessary to allow the modelling of aspect oriented constructs (aspect, advice, pointcut, etc.) in the model repository. For this purpose the VIDE/UML metamodel was extended by using the UML Profile technology. The UML Profiles (AO Profile, JPS Profile), which are required to allow modelling the AO constructs, are described in subsection 4.2.

Moreover, the horizontal model composition was chosen to process the aspect weaving. The model-to-model transformations, which are required to merge the base and aspect model and to produce the woven object oriented output model, are described in section 4.3.

In the VIDE context, the model at PIM level is not intended to be created manually, but rather to be produced by several editors, which support the VIDE PIM language. Two kinds of editors for the support of a textual and a visual syntax are planned in the VIDE project. Section 4.4 presents a proposal for extending the textual and visual syntax to integrate aspect-oriented constructs. Furthermore, to produce VIDE PIM language model from a concrete syntax, several mappings are required. This topic is discussed as an open issue at the end of this section.

4.2 AO UML Profiles

The provided UML Profiles for extending the VIDE PIM language were already partially described in D3.1. This section gives a structured overview of the completed UML Profiles, the contained elements and the possibility for modelling aspect-oriented constructs.

4.2.1 Aspect-Oriented Modelling Profile

The AO Profile is depicted in Figure 24 and contains elements, which allow modelling of aspect-oriented constructs at PIM level.

Aspect

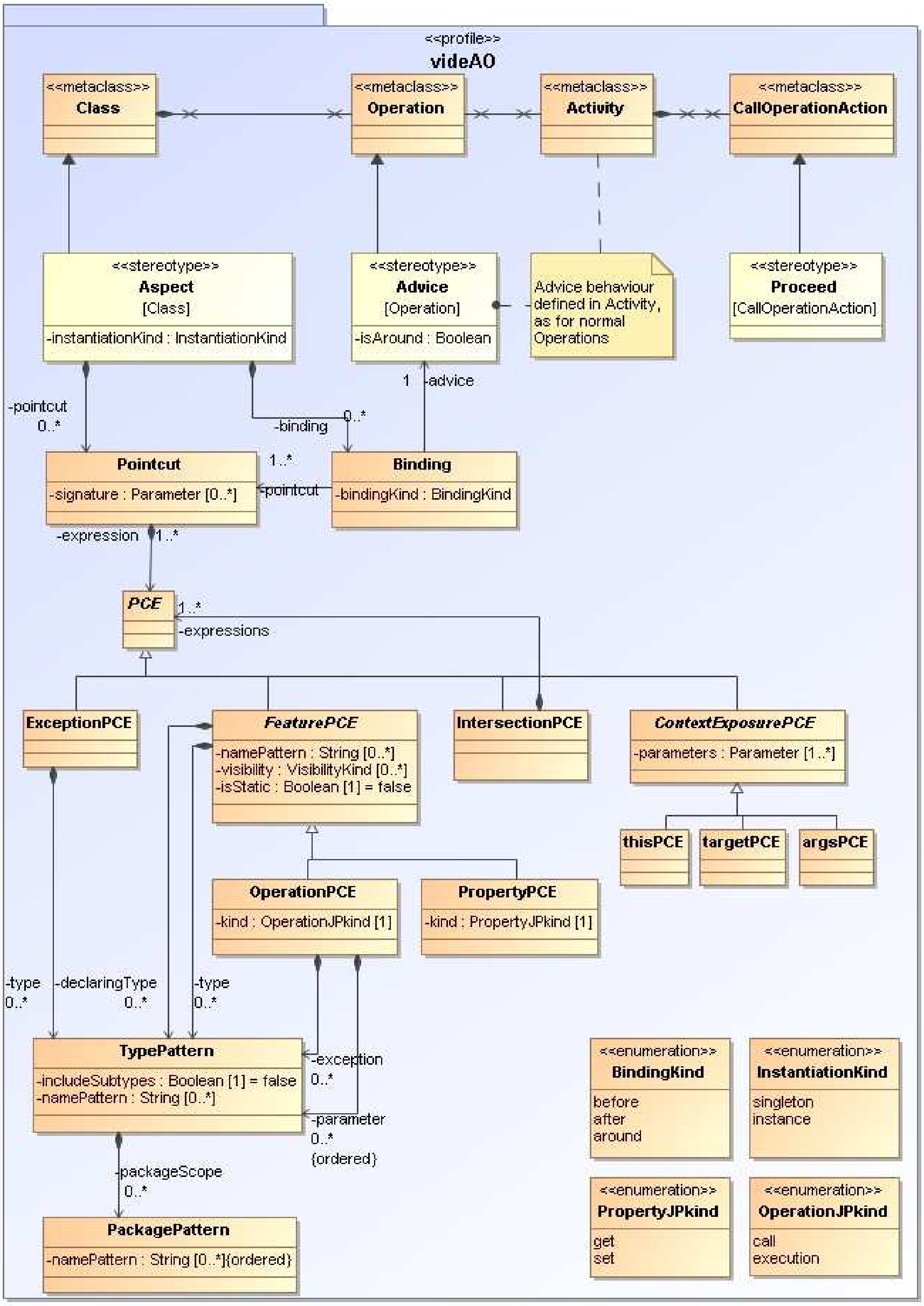
The stereotype *Aspect* is applicable to classes and represents an aspect module, which serves as a container for additional aspect-oriented (advices, bindings, pointcuts) and object-oriented (methods, etc.) constructs. The attribute *instantiationKind* defines the instantiation strategy of the aspect. The two kinds of instantiation *singleton* (one aspect instance) and *instance* (one aspect instance per class instance) are supported and covered in the enumeration *instantiationKind*.

Advice

The stereotype *Advice* is applicable to operations defined in an aspect module. This stereotype marks an operation as an advice, which contains the advice behaviour. The advice parameters are used to pass context information to the advice. The usage together with the pointcut parameters is explained in the following sections.

Proceed

The stereotype *Proceed* is applicable to CallOperationActions within an advice. This special action calls the bound joinpoint. Advices, which use the *Proceed* action, can only be bound using the binding kind "around". All parameter has to be passed to the *Proceed* action. Naturally, the target pin of the CallOperationAction must not be set.



<<profile>>
videAO
<<metaclass>> Class
<<metaclass>> Operation
<<metaclass>> Activity
<<metaclass>> CallOperationAction
<<stereotype>> Aspect [Class]
-instantiationKind : InstantiationKind
<<stereotype>> Advice [Operation]
-isAround : Boolean
Advice behaviour defined in Activity, as for normal Operations
<<stereotype>> Proceed [CallOperationAction]
-pointcut 0..*
-binding 0..*
1 -advice
Pointcut
-signature : Parameter [0..*]
1..* -pointcut
Binding
-bindingKind : BindingKind
-expression 1..*
PCE
1..* -expressions
ExceptionPCE
FeaturePCE
-namePattern : String [0..*]
-visibility : VisibilityKind [0..*]
-isStatic : Boolean [1] = false
IntersectionPCE
ContextExposurePCE
-parameters : Parameter [1..*]
thisPCE
targetPCE
argsPCE
OperationPCE
-kind : OperationJPkind [1]
PropertyPCE
-kind : PropertyJPkind [1]
-type 0..*
-declaringType 0..*
-type 0..*
TypePattern
-includeSubtypes : Boolean [1] = false
-namePattern : String [0..*]
-exception 0..*
-parameter 0..* {ordered}
-packageScope 0..*
PackagePattern
-namePattern : String [0..*]{ordered}
<<enumeration>> BindingKind
before
after
around
<<enumeration>> InstantiationKind
singleton
instance
<<enumeration>> PropertyJPkind
get
set
<<enumeration>> OperationJPkind
call
execution

Figure 24: AO Profile

Pointcut

The metaclass *Pointcut* allows modelling pointcuts, which are used to describe a set of joinpoints in the behaviour model. The pointcut does not contain a string representation of this description, but rather uses fine granular instances of pointcut expressions (see PCE) to express the pointcut definition. The resulting set of joinpoints is the union of the results of each contained pointcut expression. Furthermore the pointcut contains a signature, which is used to pass the context information to an advice. The signature of the bound advice has to be equal to the pointcut signature. The parameters, which are associated to the signature have also to be referenced by the corresponding pointcut expressions (see ContextExposurePCE). This mechanism allows the explicit assignment of context information to the parameters of the poincut.

Binding

The *Binding* class is used to associate a pointcut with an advice. The bound advice adapts the joinpoints, which are referenced by the connected pointcut. The binding also defines the binding kind (before, after, around), which is defined in the enumeration *BindingKind*. The advice is not directly associated with the pointcut, because the explicit definition allows to create n:m associations with a separate binding kind for each association.

PCE

PCE is an abstract class, which allows the modelling of pointcut expressions. Subclasses of the PCE class are used to specify the concrete properties and the kind of joinpoints, which should be captured by the corresponding pointcut.

FeaturePCE

FeaturePCE defines a feature of a class. Operations and class properties are supported. *FeaturePCE* provides different attributes and associations for specifying detailed properties of the features to be captured (namePattern, visibility, isStatic, declaringType and type). The association *type* defines the return type if operation is specified. Otherwise the association *type* defines the type of the specified field.

OperationPCE / PropertyPCE

OperationPCE defines joinpoints, which relate to an operation. To distinguish between the two different joinpoint kinds (call of an operation (CallOperationAction) and the execution of an operation (Operation)) the attribute *kind* has to be set. Possible values are defined in the enumeration *OperationJPKind*. In an analogous way, the *PropertyPCE* defines joinpoints, which corresponds to a property of a class. The values, defined in the enumeration *PropertyJPKind* can be set to the property *kind*.

ContextExposurePCE

The *ContextExposurePCEs* are responsible for selecting joinpoints with the specified context and also for passing the context information to the bound advice. Context informations are for instance arguments of an operation call and value, which is set to a property.

For this purpose, the *ContextExposurePCEs* contains a list of parameters. If the captured context of the current pointcut expression should be passed to the advice, the corresponding instance of the Parameter defined in the pointcut signature has to be referenced by the parameter list of the pointcut expression. The *ContextExposurePCEs* can also be used to filter the resulting set of joinpoints without passing the context information to the advice. In this case, the parameter referenced by the pointcut expression must not be referenced by the pointcut signature.

ThisPCE

The *ThisPCE* captures a joinpoint, if the object, where the joinpoint is triggered, is an instance of the type (or subtype of the type) of the first parameter defined in the parameter list.

TargetPCE

The *TargetPCE* captures a joinpoint, if the target object, on which the joinpoint is triggered, is an instance of the type (or subtype of the type) of the first parameter defined in the parameter list.

ArgsPCE

The *ArgsPCE* captures a joinpoint, if the arguments (call/execution joinpoints) or the value to be get/set from/to a property match to the parameter list associated by the *ArgsPCE.*

IntersectionPCE

The IntersectionPCE allows combining pointcut expressions to limit the resulting set of joinpoints.

TypePattern / PackagePattern

Instances of these metaclasses are used to express types and packages. Within the *namePattern*, wildcards can be used to select related elements. Additionally the TypePattern can be enabled for covering subtypes by setting the property *includeSubtypes*.

4.2.2 Join Point Shadowing Profile

The JPS Profile depicted in Figure 25 provides additional stereotypes to annotate resolved joinpoints in the base model. Supported joinpoints are:

- Call (stereotype CallJPShadow)
- Execution (stereotype ExecutionJPShadow)
- PropertySet (stereotype PropertySetJPShadow)
- PropertyGet (stereotype PropertyGetJPShadow)

These additional stereotypes are defined in a separate JPS Profile to facilitate an uncomplicated extension. If we want to consider the OCL expressions, which are mainly used in the VIDE PIM language for evaluating properties (e.g. PropertyCallExp), it is only necessary to change the stereotype PropertyGetJPShadow to be applicable to instances of the metaclass PropertyCallExp.

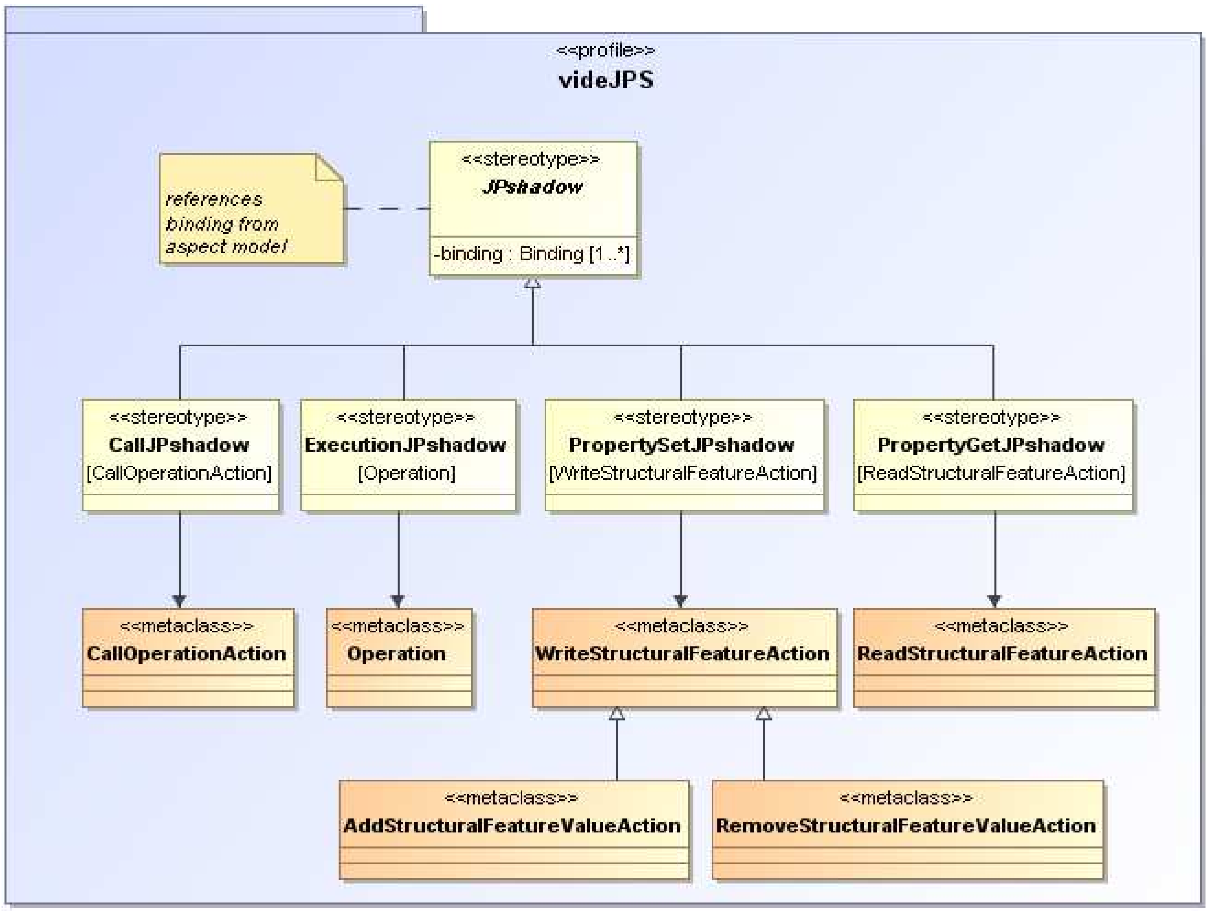


Figure 25: JPS Profile

4.3 Specification of Model Transformations for AO Composition

As already described in D3.1 and also shown in Figure 26, the proposed aspect composition process is separated in two phases (pointcut resolving and aspect composition) and requires two input models (base model and aspect model). Nevertheless, each of the mentioned phases can consist of more than one transformation iteration with several intermediate models. The section 4.3.1 (Pointcut Resolving) and section 4.3.2 (Aspect Composition) give a detailed description of the core transformations, which are required to realize our approach. The structure of the description is similar to the template for the description of ATL transformations (which can be found at the ATL website [18]), where the usage of semi formal as well as textual description (pseudo code) of transformation rules is allowed.

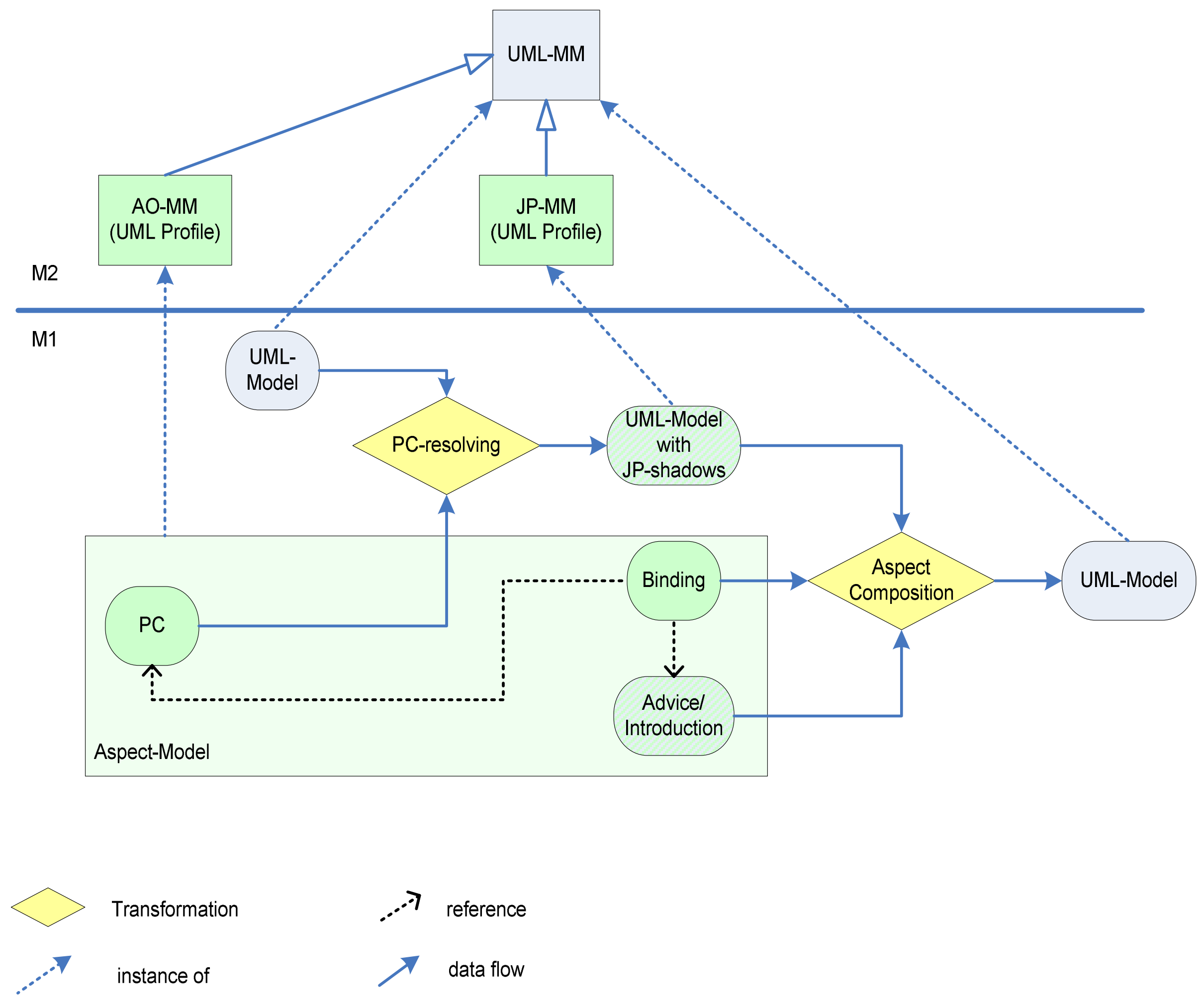


Figure 26: Aspect Oriented Composition

The transformation description is structured as follows:

- Transformation
    - Description
        - Textual description of the transformation
    - Inputmodel(s)
    - Outputmodel(s)
    - Rules
        - Rules, which are required to process the transformation
        - Not exhaustive (similar rules are not fully described, open issues, etc.)
    - Operations
        - Helper operations, to reduce complexity of rules

The rules are structured as follows:

- Rule
    - Description
        - Textual description
    - FromElement
        - Element in the input model, on which the rule is triggered
    - Precondition
        - Precondition for the execution of the current rule
    - Actions
        - Actions, which are part of the rule and have to be processed.
        - To clarify the intent of the described rule, pseudo code similar to ATL code [18] as well as textual phrases are used in the description.

Elements, which serve not as a starting point for a described rule or where the precondition is not fulfilled, are copied from the source model(s) to the target model(s)

After the core transformations are described, this section gives an overview of the additional features, e.g. the handling of OCL expressions and of dynamic joinpoint properties during static weaving at PIM level.

4.3.1 Pointcut Resolving

Transformation: Pointcut Resolving

Description: Searches for and annotates model elements, which are selected by a pointcut

Inputmodel: Aspect Model (VIDE/UML2 + AO Profile), Base Model (VIDE/UML2)

Outputmodel: Intermediate Model (VIDE/UML2 + JPS Profile)

Rule: ExecJPS

Description: Assigns a corresponding ExecutionJPshadow stereotype to an Operation, which is matched by a pointcut.

FromElem: `op: uml::Operation`

Precondition: `isJoinpoint(op)`

Actions:

1. Create `opTarget: uml::Operation`
2. Copy all properties of `op` to `opTarget`
3. Create `execJPS: JPSProfile::ExecutionJPshadow` with following property assignment
    a. `base_Operation <- op`
    b. `binding <- getBindings(op)`
4. Assign `execJPS` to `opTarget`

Rule: CallJPS

Description: Assigns a corresponding CallJPshadow stereotype to a CallOperationAction, which is matched by a pointcut.

FromElem: `callOp: uml::CallOperationAction`

Precondition: `isJoinpoint(callOp)`

Actions:

1. Create `callOpTarget: uml::CallOperationAction`
2. Copy all properties of `callOp` to `callOpTarget`
3. Create `callJPS: JPSProfile::CallJPshadow` with following property assignment
    a. `base_Operation <- op`
    b. `binding <- getBindings(op)`
4. Assign `callJPS` to `callOpTarget`

Rules for the remaining joinpoint shadow kinds can be structured in an analogous way.

Operation: isJoinpointShadow

Description: Detects if a model element is matched by any pointcut.

Parameter: Potential joinpoint shadow

Return: Boolean

isJoinpointShadow(`potJPShadow`)

1. return `getBindings(potJPShadow).notEmpty()`

Operation: getBindings

Description: Detects all bindings, whose associated pointcut matches the potential joinpoint shadow.

Parameter: Potential joinpoint shadow

Return: Sequence of bindings

getBindings(`potJPShadow`)

1. return all bindings `b`, where `b.pointcut` matches `potJPShadow`

Operation: match

Description: Detects, if an element is matched by a pointcut

Parameter: Potential Joinpoint shadow, Pointcut

Return: boolean

match(`potJPShadow, pointcut`)

1. return true, if at least one of the associated pointcut expressions (`pointcut.expression`) matches `potJPShadow`
    a. `pce: JPSProfile::OperationPCE` matches `potJPShadow` if:
        i. kindOf(`potJPShadow`) = = `pce.kind` and
        ii. name(`potJPShadow`) is covered by `pce.namePattern` and
        iii. visibility(`potJPShadow`) = = `pce.visibility` and
        iv. isStatic(`potJPShadow`) = = `pce.isStatic` and
        v. analogous for the remaining attributes and associations of `pce`
    b. `pce: JPSProfile::PropertyPCE` matches `potJPShadow` if:
        i. analogous to OperationPCE
    c. `pce: JPSProfile::IntersectionPCE` matches `potJPShadow` if:
        i. all associated pointcut expressions (`pce.expressions`) match `potJPShadow`
    d. `pce: JPSProfile::ThisPCE` matches `potJPShadow` if:
        i. the instance , where `potJPShadow` is triggered is instance of `pce.parameters[0].type`
    e. `pce: JPSProfile::TargetPCE` matches `potJPShadow` if:
        i. the instance, on which `potJPShadow` is triggered is instance of `pce.parameters[0].type`
    f. `pce: JPSProfile::ArgsPCE` matches `potJPShadow` if:
        i. the arguments (or value to get or to set) of `potJPShadow` are instances of the types of parameters defined in `pce`

4.3.2 Aspect Composition

The aspect composition transformations are more complex, thus this section focuses on some core transformations, which can be modified and extended to support special features, as will be described in later sections.

Transformation: **JPS Extraction**

Description:

This transformation extracts several joinpoint shadows to a separate activity. The goal of this transformation is the provision of explicit access to the control flow, which is required by the weaving transformations. Using UML Action semantics, in some cases it is possible to model only the object flow, but the proposed aspect composition adapts the behaviour by changing the control flow. This transformation does not change the observable behaviour, but rather prepares the behaviour model for processing the aspect composition.

Of course, if the behaviour is modelled using explicit control flow, the transformation has not to be processed. The processing of this transformation is meaningful for the following joinpoint shadow kinds:

- CallJPshadow
- PropertySetJPshadow
- PropertyGetJPshadow (if OCL expressions are used for reading/evaluating a property, this transformation cannot be processed, because the joinpoint shadow can be placed within a nested OCL expression)

For each supported joinpoint shadow a separate rule is required. Because of similarity, only the rule for extracting call joinpoint shadows is described.

Inputmodel: Intermediate Model (VIDE/UML2 + JPS Profile)

Outputmodel: Intermediate Model (VIDE/UML2 + JPS Profile)

Rule: ExtractCallJPshadow

Description: Extracts call operation joinpoint shadows to a separate activity and connects the control and object flows. The original join point shadow is replaced by a call of the created activity. The behaviour is not changed.

FromElement: `callOp: uml::CallOperationAction`

Precondition: `hasJPSStereotype(op)`

Actions:

1. Create `act: uml::Activity` in parent class of `callOp`
2. Create `initNode: uml::ActivityInitialNode` in `act`
3. Create `finalNode: uml::ActivityFinalNode` in `act`
4. Create `beforeCF: uml::ControlFlow` in `act`
5. Create `afterCF: uml::ControlFlow` in `act`
6. Create `targetCallOp: uml::CallOperationAction` in `act`
    a. Copy all relevant properties of `callOp` (operation, etc.)
    b. Copy stereotype marking `targetCallOp` as a join point shadow
7. Connect `initNode` and `targetCallOp` using `beforeCF`
8. Connect `finalNode` and `targetCallOp` using `afterCF`
9. For each InputPin `curInputPin` in `callOp` do
    a. Create `inputParam: uml::ActivityParameterNode` in `act`
        i. `inputParam.Type <- curInputPin.Type`
    b. Create `inPin: uml::InputPin` in `targetCallOp`
        i. `inPin.Type <- curInputPin.Type`
    c. Create `inOF: uml::ObjectFlow`
    d. Connect `inputParam` and `inPin` using `inOF`
10. For each OutputPin `curOutputPin` in `callOp` do
    a. Create `outputParam: uml::ActivityParameterNode` in `act`
        i. `outputParam.Type <- curoutputPin.Type`
    b. Create `outPin: uml::OutputPin` in `targetCallOp`

            i. `outPin.Type <- curOutputPin.Type`
    c. Create `outOF: uml::ObjectFlow`
    d. Connect `outputParam` and `outPin` using `outOF`
11. Create `callBeh: uml::CallBehaviourAction`
    a. Copy all input and output pins of `callOp`
    b. Connect control flow
        i. `callBeh.Incomming <- callOp.Incomming`
        ii. `callBeh.Outcomming <- callOp.Outcomming`
        iii. `callOp.Incoming.target <- callBeh`
        iv. `callOp.Outcomming.source <- callBeh`

Transformation: **Advice Weaving**

Description:

This transformation creates the required model infrastructure (aspect classes, required interfaces, etc.), adapts the control flow around the joinpoint shadows and connects the context information with the corresponding advice. A description of transformation rules will be provided for the CallJPshadow and the binding kind *before* and *around*. Other permutations are handled in an analogous way.

Inputmodel: Intermediate Model (VIDE/UML2 + JPS Profile)

Outputmodel: Woven Model(VIDE/UML2)

Rule: InfrastructureCreation

Description:

This rule creates the following required model infrastructure:

- aspect class with the defined instantiation strategy
- advice operation (incl. transformation of the Proceed action)
- Interface for closure classes

Note: If an operation/parameter is created/copied/modified, also the associated activity/activity parameter is created/copied/modified.

FromElement: `aspect: AOProfile::Aspect`

Precondition: none

Actions:

1. Create new class for the current aspect
2. Copy all operations from aspect to the created class
3. For each advice operation do:
    a. Create an interface AroundClosure:
        i. Set an individual name, because for each advice an AroundClosure interface is created

    ii. Create a public operation runProceed with the same return type and the same signature like one of the bound joinpoint shadows

  b. Create an operation with the advice signature in the newly created class.

    i. Add a parameter with the type of the AroundClosure interface to the signature

  c. Copy the advice behaviour to the created activity.

  d. Replace the Proceed action with the call to the runProceed method of the new parameter, which has the type of the AroundClosure interface.

    i. Reconnect the control and object flows

4. Create a static method `getAspectOf(Object o)` in the aspect class. This method is later used to get the aspect class instance. Since the approach supports two instantiation strategies, two mechanisms for creating an instance of the aspect class have to be supported (see next steps).
5. If `aspect.instantiationKind == singleton`
    a. Create a static field in the aspect class to store the single aspect class instance.
    b. Create the behaviour for creation of the singleton instance (see Singleton design pattern)
6. If `aspect.instantiationKind == perThis`
    a. Create a static field, storing a hash table to store associations between an object and the corresponding aspect class instance
    b. Create the behaviour for creating and managing aspect class instances

Rule: BehaviorAdaptationCallJPSBefore

Description:

This rule inserts additional behaviour before a captured operation call.

FromElement: `callJPS: JPSProfile::CallJPShadow`

Precondition: `callPS.binding[0].bindingKind == before` (The approach only supports weaving of one bound advice per joinpoint shadow, see section 4.3.4)

Actions:

1. Create `getAsp: UML::CallOperationAction` to call the static method `getAspectOf(Object o)` of the aspect class, which contains the bound advice
2. Create `callAdvice: UML::CallOperationAction` to call the advice method on the corresponding aspect class instance
3. Connect the output of `getAsp` with the target input pin of `callAdvice`
4. Reconnect the control flow to achieve the following calling order: `getAsp, callAdvice, callJPS`
5. Assign context information

a. Analyze corresponding pointcut for context exposure pointcut expressions. If a context exposure pointcut expression references a parameter from the pointcut signature, assign the corresponding value to the corresponding parameter in the advice call (note, pointcuts and advices have to have the same signature, otherwise they cannot be bound)
    i. *ThisPCE*
        1. Assign the self object value getting by the ReadSelfAction to the corresponding advice call parameter by creating a new object flow between the output pin of the ReadSelfAction and the parameter.
    ii. *TargetPCE*
        1. Assign the object, on which the captured joinpoint shadow is called to the corresponding advice call parameter by creating a new object flow between the relevant object and the parameter.
    iii. *ArgsPCE*
        1. Assign the parameters which are passed to the captured call to the corresponding parameters of the call to the advice. Use object flow between the value sources and the targets (corresponding input pins of the advice call).

Rule: BehaviorAdaptationCallJPSAround

Description:

This rule inserts additional behaviour around a captured operation call.

FromElement: `callJPS: JPSProfile::CallJPShadow`

Precondition: `callPS.binding[0].bindingKind == around` (The approach only supports weaving of one bound advice per joinpoint shadow, see section 4.3.4)

Actions:

1. Extract captured CallOperationAction to a separate public Operation `extrOp: uml::Operation` (respective to the assigned activity). This step is necessary to allow the advice to call also private operations.
    a. Create new public Operation
    b. Set an unique name
    c. Copy the captured CallOperationAction to the created Operation
    d. Reconnect required object and control flows.
2. Create an individual Closure class. A concrete instance of this class will later be passed to the advice, where the interface of the Closure object can be used to call the captured joinpoint shadow using a standard interface (runProceed method in interface AroundClosure)
    a. Create a new Closure class, which implements the AroundClosure interface, which was created in the context of the bound advice.
    b. The newly created class contains:

i. Field `target`. The type is the declaring type of `extrOp.` This field is set by the constructor

ii. Operation `runProceed()` which is an implementation of the operation `runProceed()` declared in the AroundClosure interface (same signature and return type) .

   1. The operation contains a call to `extrOp` on `target.` All parameters are connected using the object flow.
   2. If the `runProceed()` Operation returns a value, the value of `target.extrOp` is returned.

3. Create an Action `createClosure: uml::CreateObjectAction` to get an instance of the Closure class.
4. Connect the self object (e.g. available by the ReadSelfAction) to the input pin of `createClosure`.
5. Create `getAsp: UML::CallOperationAction` to call the static method `getAspectOf(Object o)` of the aspect class, which contains the bound advice
6. Create `callAdvice: UML::CallOperationAction` to call the advice method on the corresponding aspect class instance
7. Replace `callJPS` with `callAdvice` and reconnect existing control flow.
8. Connect the output of `getAsp` with the target input pin of `callAdvice`
9. Reconnect the control flow to achieve the following calling order: `createClosure`, `getAsp, callAdvice`
10. Assign context information
    a. Analyze corresponding pointcut for context exposure expressions. If a context exposure pointcut expression references a parameter from the pointcut signature, assign the corresponding value to the corresponding parameter in the advice call (note, pointcuts and advices have to have the same signature. Otherwise they cannot be bound)

       i. *ThisPCE*

          1. Assign the self object value with the help of the ReadSelfAction to the corresponding advice call parameter by creating a new object flow between the output pin of the ReadSelfAction and the parameter.

       ii. *TargetPCE*

          1. Assign the object, on which the captured joinpoint shadow is called to the corresponding advice call parameter by creating a new object flow between the relevant object and the parameter.

       iii. *ArgsPCE*

          1. Assign the parameters which are passed to the captured call to the corresponding parameters of the call to the advice. Use object flow between the value sources and the targets (corresponding input pins of the advice call)

11. Connect the output pin of `createClosure` to the corresponding parameter of `callAdvice` using an object flow.

4.3.3 Additional AO composition features

This section describes additional features of the aspect composition which are relevant or could be useful in the VIDE context.

4.3.3.1 Handling of OCL expressions

In the VIDE PIM language, OCL expressions (e.g. FeatureCallExp) are used to evaluate/read features instead of actions from UML Action Semantics (e.g. ReadStructuralFeatureAction).

Supported OCL expressions are described in the VIDE metamodel and are stored in the model repository as metamodel instances and not as one generic instance with a textual description of the OCL expression. The aspect composition phase “Pointcut Resolving” is not affected by the usage of OCL expressions, because it is possible to search for metamodel instances with a specific type. The corresponding transformations can be adapted easily.

As already mentioned, the “Aspect Composition” requires explicit control flow for the behaviour adaptation. If a single OCL expression is identified and marked as joinpoint shadow, the aspect composition can take place in the described way. No conceptual changes are required.

However, the usage of nested OCL expressions (e.g. see figure 27) causes problems, because no explicit control flow is modelled within the OCL expression. Only the modelled control flow of the root expression is accessible by the transformations.

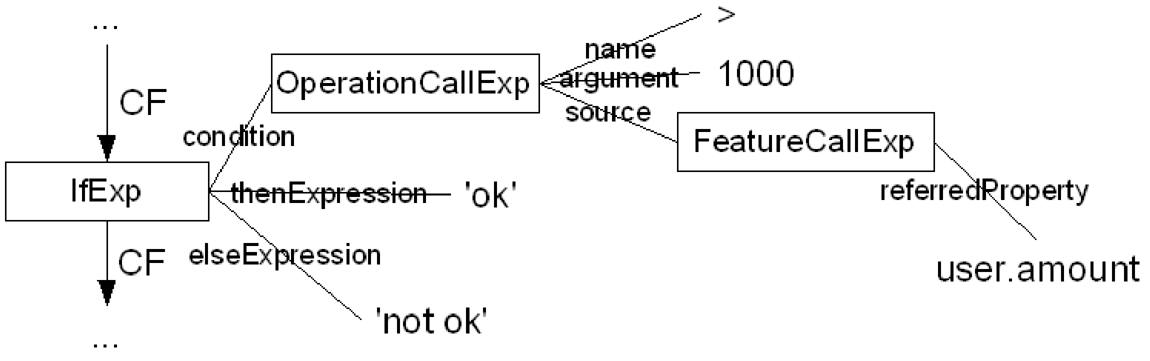


Figure 27: Example of a nested OCL expression

If a nested OCL expression was identified as a joinpoint shadow (e.g. the FeatureCallExp in Figure 27), it is not possible to directly adapt the behaviour, represented by the identified joinpoint shadow.

Following solutions were researched:

- Additional model-to-model transformations could translate the OCL expressions into UML action semantics using explicit control flow and process the aspect composition in the described way. This solution is not applicable, because the metamodel description of the VIDE/UML language provides a subset of the UML metamodel, where UML actions are not intended to be used for reading and evaluating properties.

- If we want to weave the additional behaviour directly into the nested OCL expressions, there is a possibility to call user defined operations from OCL expressions. Before the execution of the OCL expressions by a special engine, the expressions can be rewritten for instance to optimize performance, which can cause inconsistency in the woven behaviour.
- The pragmatic approach for solving this problem is to process the aspect weaving in the common way on the root OCL expression, which provides access to the control flow. This strategy could cause the effect of "imprecise" weaving, but the opportunity scenario has shown that this solution is the most suitable for modelling and composing all analyzed crosscutting concerns.

4.3.3.2 Handling of dynamic join point properties during static weaving

The described approach processes static pointcut resolving only. After the pointcut resolving, all joinpoint shadows are determined.

Especially the static type checking during the resolving of the context exposure pointcut expressions can cause the effect, that some special elements are wrongly not determined as joinpoint shadows (e.g. because of polymorphism).

During a static analysis the dynamic properties (runtime properties) are not present or had to be statically approximated using an expensive analysis. One possible solution is to statically determine only potential joinpoint shadows and to weave dynamic conditions before the advice execution. Whether a conditional advice will be executed, is decided at runtime.

4.3.3.3 Proceed action without a complete signature

In some cases there is no need to modify values, which are passed to the *proceed* action in an advice. The possibility for calling the *proceed* action without the complete signature can reduce modelling effort and could provide a more general and flexible advice model. The described approach can be adapted for providing this feature by buffering the parameters in the Closure object. In this case, the parameters have not to be passed to the advice and to the proceed call. Rather they are passed "automatically" to the *proceed* call.

4.3.3.4 Joinpoint Reflection

Several advices, which model crosscutting concerns such as debugging and profiling, require the facility, to get information about the joinpoint (operation name, field name, etc.), which causes the advice call. The AspectJ approach [21] provides a generic class JoinPoint, which contains several attributes for representing the joinpoint´s context information. During the aspect weaving, the advice signature is extended by adding a parameter of type JoinPoint. Before the advice is called, a concrete instance of the class JoinPoint is created and filled with the context information. This instance is passed to the advice, which can access the context information.

This mechanism can also be integrated in the developed aspect composition approach.

4.3.4 Open issues

The following areas in the domain of aspect orientation are not considered in the described approach:

- **Advice precedence**: The described AO Profile is designed to support more than one advice to be bound to a joinpoint, but there is no possibility to define, in which order advices should be considered during the aspect composition. The approach of Fuentes and Sanchez [15] uses a kind of prioritisation of advices by adding an integer value to each advice. Furthermore there are a lot of approaches described (e.g. [16]), which show the complexity of the problem.
- **Structural introduction:** Structural introductions are not considered in the described approach, which allow focusing on the behaviour adaptation on PIM level.
- **Validation of base and aspect models:** A suitable validation of the input models can support the modeller and detect errors already during the modelling phase. A validation requires a complex analysis, which goes beyond the scope of this deliverable.

4.4 VIDE Syntax extension for aspect-oriented constructs

If we want to enable the editors, which are intended to be developed in the VIDE project, to produce VIDE PIM language with the aspect oriented extension, it is also required to support the definition of aspect oriented constructs.

Two kinds of editors are planed to be developed. The textual editor allows to use the textual syntax of the VIDE language, whereas the visual editor uses a visual syntax, to describe VIDE programs.

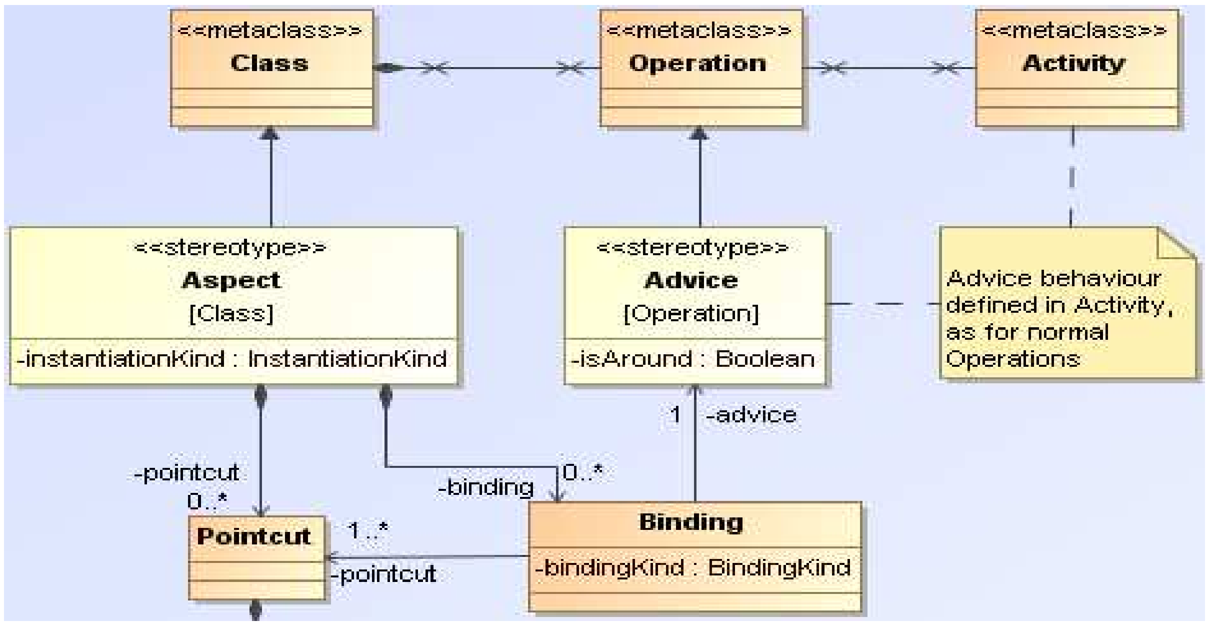


Figure 28: Relevant elements to be covered by a syntax extension

Not all aspect-oriented constructs, which are defined in the AO Profiles have to be supported explicitly in the VIDE syntaxes. With respect to the relevant part of the AO Profile (see Figure 28), the following constructs were identified to be supported by the concrete VIDE syntaxes:

- Aspects, to provide a new kind of module for encapsulating crosscutting concerns
- Advices, to encapsulate behaviour, which has to be inserted at the identified joinpoints
- Pointcuts, to declare elements on the modelled behaviour to be adapted.
- Bindings, to associate pointcuts with advices to be bound.

The behaviour in an advice can be expressed by using the already defined constructs for describing behaviour in the textual and visual syntax of the VIDE language.

The following sections give an overview of the proposals for the extensions of textual and visual syntaxes. This proposals are example-based and do not provide formal extensions such as an EBNF extension of the VIDE language.

4.4.1 Textual Syntax

The proposal, described in this section, allows describing aspects as superior modules, which can contains advices, pointcuts, advice operations but also common operations and fields, etc. Therefore, an aspect should be an extension of a class. An overview of the proposed syntax is depicted in Listing 1.

```
aspect Foo
{
    pointcut foo(int i, String s)
    {
       call (* bar(int i, String s)
    }

    advice fixFoo(int i, String s): String : around foo(i, s)
    {
        //... do something
        return proceed();
    }
}
```

Listing 1: Syntax overview

A pointcut is similar to a method and contains also a signature. However in the pointcut´s body only declarative parts can be used, no semicolon is required after the pointcut definition. To specify the pointcut definition, pointcut expressions can be defined within a pointcut body. The defined pointcut expressions are combined with an implicit OR-relation (","). An example is shown in Listing 2.

```
pointcut foo(int i, String s)
{
    call /* .. */, execution /* .. */, get /* .. */, set /* .. */
}
```

Listing 2: Combined pointcut expressions

The combination of the advice signature and pointcut signature during the binding allows for passing context information determined during the pointcut resolving into the advice. A Pointcut is bound to an advice using one of the following binding specifiers: around, before, after (See Listing 3). Listing 3 also depicts the usage of the Proceed keyword, which allows calling the captured joinpoint from an advice.

```
advice fixFoo(int i, String s): String : around foo(i, s)
{
    return proceed();
}
```

Listing 3: Advice binding

The proposed syntax does of course not support all features, which are able to be expressed in the model repository using the AO Profile extension. The main goal of a concrete textual syntax should aim at the specific requirements in the used domain.

4.4.2 Visual Syntax

The proposal for the visual syntax, which is presented in this section, is based on the work of Han, Kniesel and Cremers [17].

Aspects are visualized similar to classes. Additionally aspects contain advices, pointcuts and pointcut expressions (see Figure 29).

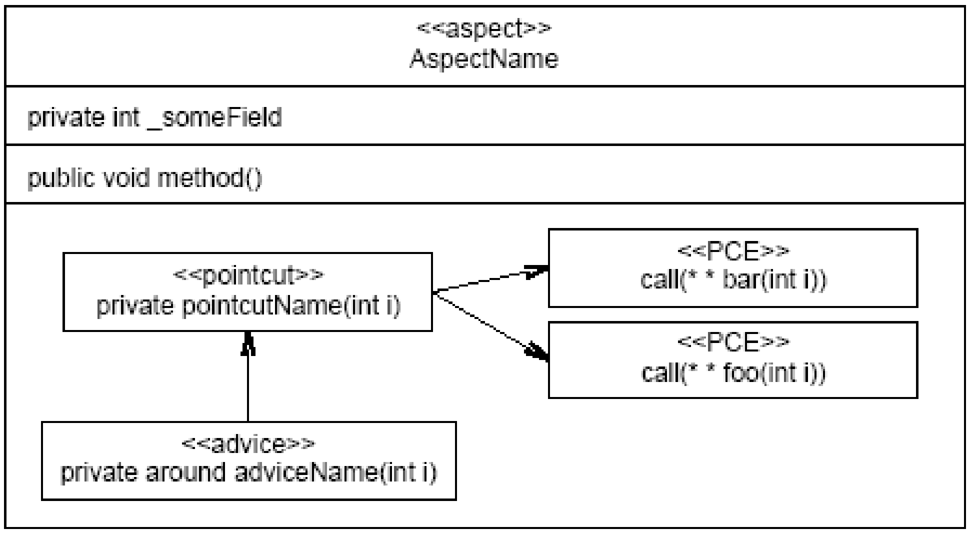


Figure 29: Proposal for visual syntax of aspect oriented constructs

The binding is realized by an association between the pointcut and the corresponding advice. The pointcut is associated to the declaring pointcut expressions, which contains a textual description. This variant increases the usability of the syntax, because to model all pointcut expressions by using detailed instances of required classes would cause more effort.

The mapping between the textual and visual syntaxes is responsible for transforming the textual pointcut definition into the corresponding metamodel instances of the VIDE PIM language.

4.4.3 Open issues

To allow a flexible description of pointcuts a pointcut language is required. For this purpose a new domain-specific language could be designed. To process and transform the pointcut language constructs a parser is required. Therefore it could be more efficient to choose one of the existing approaches (e.g. the pointcut language of AspectJ).

As already described, both proposals are designed for using pointcut declarations in a textual syntax, so the process for parsing and transforming the pointcut definition can be reused.

Furthermore, to transform the visual and textual syntax into the VIDE model repository, several mapping transformations are required.

5 Summary and Conclusions

Both Aspect-Oriented Software Development (AOSD) and Model Driven Development (MDD) are approaches to reduce complexity in software development. These approaches use different but complementary ideas to reduce complexity. AOSD adds additional modules and a weaving mechanism to extract tangled and scattered functionality, so called crosscutting concerns. MDD reduces complexity by replacing the writing of source code by using abstract models instead; executable code is generated from the models. Crosscutting concerns appear already in the modelling phases of MDD while aspect-oriented programs can have a lot of source code. So it seems natural that a combination of the two approaches can have the advantages of both and thus can help overcome complexity in software development. Task 3.3 was aimed at evaluating the developed aspect-oriented composition approach and at providing a specification of AOC to be supported by the VIDE project.

In Section 2, we reviewed and evaluated our approach, which was developed in the first period of WP3. Some deficiencies were identified. Due to the flexibility of the selected AO Modelling approach, the required extension to the existing AO Profiles has been done. With the extended AO Profile, a suitable modelling of identified crosscutting concerns has been made possible.

To show, where our approach is settled and compare it to other approaches, different variations of aspect-oriented composition were discussed at several levels. This discussion shows the flexibility of our approach (e.g. supporting different instantiation strategies and the extensibility of AO Profiles).

In the last part of section 2, the chosen evaluation criteria for the developed AO modelling approach are presented and associated with suitable metrics. The metrics help, to show the impact of the usage of our AO modelling approach on the selected evaluation criteria.

Section 3 presented a review of the business scenario, which was already presented in D3.1. Furthermore, the models of the crosscutting concerns, required for the empirical evaluation, were described. This description contains the object-oriented models as well as the aspect-oriented models of the same crosscutting concerns. This allowed us to apply the selected metrics on both variants and compare them with respect to understandability and maintainability. The results have shown that the maintainability of the crosscutting concern functionality was improved by using our AO modelling approach. The complexity of methods, where the crosscutting concern was extracted from, was reduced, which leads to an improved understandability. On the other hand, the aspect-oriented modelling approach has introduced an additional implicit coupling between advices and joinpoints. This effect decreases the understandability. However, the usage of suitable tool support (e.g. static analysis for pointcut resolving during the modelling) can minimize this effect.

In Section 4 the specification of the AO Profiles and the required core model transformations for aspect composition were presented. For a complete realisation, additional transformations, which were not explicitly specified, are required. They can be derived from the existing transformations. Furthermore, a textual description of additional useful features, which can be integrated as an additional step/rule in certain transformations, was given. The extensions for the textual and visual syntax were not specified in a formal way. Proposals with concrete examples were presented, to show, which constructs are necessary and should be considered during the next activities of the VIDE projects.

All open issues, which were outlined in D3.1, were considered during the second period of the Work package 3. The requirements for providing suitable aspect models of the crosscutting concern Consistency Check were analyzed (not described explicitly in D3.2) and accordingly to these requirements, the AO Profiles were extended to suit the needs.

Also the facility for modelling the same behaviour in different ways (control flow based, object flow based) were researched. As a result, an intermediate transformation was added to the aspect composition. This transformation extracts the joinpoint shadows to a separate activity, where the explicit control flow can be accessed by the following composition transformations.

Section 4 provided proposals for the syntax extensions of the textual and visual VIDE syntax. These proposals are not sufficient for the realisation within the VIDE prototypes. Only a short overview of required constructs and possible representations was given. Formal extensions e.g. EBNF of the textual syntax extension should be defined before integrating the syntax extension in the VIDE prototype (see section 5.2).

Moreover, the impact of the usage of OCL expressions for evaluating/reading features was analyzed and possible solutions were discussed in section 4.

5.1 Demonstrator

Parts of the developed concepts were realized as a Demonstrator, to show their feasibility and suitability. This demonstrator is neither a prototype nor a tool to be used. The demonstrator consists of the following software artefacts, which are only executable in an eclipse-based environment (see README in Demonstrator.zip):

- AO Profiles
- ATL Transformations for basic concepts of Pointcut Resolving and Aspect Composition at PIM level
- Example models

The Demonstrator is not exhaustive.

The realisation showed the high complexity of the required transformations for Pointcut Resolving and Aspect Composition. Furthermore the unstable version of the ATL Eclipse plug-in caused some problems. It was for instance not easy to decide, if some of the errors, which have occurred during the development, were caused by our ATL transformations or by a bug in the ATL implementation.

5.2 Outlook

The described concepts for pointcut matching and aspect composition can be basically integrated in the PIM visual editor for VIDE that will be developed in Work package 9. The transformations presented in this deliverable can be used and extended for that purpose. For the integration of aspect-oriented constructs in the textual and visual syntax, our proposal should be used to define a formal specification, e.g. by extending the EBNF definition of the VIDE textual syntax. Also the specification of the mapping between the additional syntactical constructs should be defined. Such a mapping specification was not in the scope of this deliverable.

The Demonstrator developed in Workpackage 3 is not part of Work package 9, which deals with the development of the VIDE prototype. Nevertheless, if similar technology to the one used in the demonstrator will be chosen for the prototype, suitable artefacts of our demonstrator should be reused and/or modified.

Abbreviations

CIM	Computation Independent Model
PIM	Platform Independent Model
PSM	Platform Specific Model
MDA	Model Driven Architecture
JPS	Join Point Shadow
AO	Aspect Orientation
AOP	Aspect-Oriented Programming
AOM	Aspect-Oriented Modelling
AOC	Aspect-Oriented Composition
ATL	ATLAS Transformation Language

References

1. McCabe & Associates, *McCabe Object-Oriented Tool User's Instructions*, 1994.
2. Java Method Cyclomatic Complexity, http://www.leepoint.net/notes-java/principles_and_practices/complexity/complexity-java-method.html
3. Rosenberg and Hyatt, *Software Quality Metrics for Object-Oriented Environments*, April 1997 http://satc.gsfc.nasa.gov/support/CROSS_APR97/oocross.PDF
4. Sant' Anna et al., *On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework*. Proc. of Brazilian Symposium on Software Engineering (SBES'03), Manaus, Brazil, Oct 2003, 19--34.
5. M. L. Lee, *Change Impact Analysis of Object-Oriented Software*, PhD Thesis, George Mason University, Virginia, USA, 2000
6. Hippner, H. and Wilde, K. D. (2002). CRM—Ein Überblick, in S. Helmke, M. Uebel and W. Dangelmaier (eds), Effektives Customer Relationship Management: Instrumente—Einführungskonzepte—Organisation, second edition, Gabler, Wiesbaden, pp. 3–37.
7. Buck-Emden R., Zencke, P., mySAP *CRM: The Official Guidebook to SAP CRM Release 4.0*, SAP Press, May 2004
8. SAP AG, *SAP CRM*, http://www.sap.com/solutions/business-suite/crm/index.epx
9. SAP AG, SAP Netweaver Developer Studio, http://help.sap.com/saphelp_nw04/helpdata/en/cb/f4bc3d42f46c33e10000000a11405a/frameset.htm
10. TopCased, http://www.topcased.org/
11. SAP AG, Partner Determination Procedures, SAP Library http://help.sap.com/saphelp_crm40/helpdata/en/3c/92ecee484a11d5980800a0c9306667/content.htm
12. Khanna, A. *How to set up partner determination in mySAP CRM*, CRM Expert http://www.crmexpertonline.com/archive/Volume_03_(2007)/Issue_01_(January_and_February)/v3i1a3.cfm
13. Aspectj homepage, October 2006. http://www.eclipse.org/aspectj/.
14. AspectJ Development Tools, http://www.eclipse.org/ajdt/
15. Lidia Fuentes and Pablo Sánchez. *"Designing and Weaving Aspect-Oriented Executable UML models"*. Journal of Object Technology - Special Issue on Aspect-Oriented Modelling.
16. Jing Zhang, Thomas Cottenier, Aswin van den Berg, and Jeff Gray: "Aspect Composition in the Motorola Aspect-Oriented Modelling Weaver", in Journal of Object Technology, vol. 6, no. 7,Special Issue. Aspect-Oriented Modelling, August 2007, pp 89-108 http://www.jot.fm/issues/issue_2007_08/article4.
17. Yan Han, Günter Kniesel, Armin Cremers: Towards Visual AspectJ by a Meta Model and Modelling Notation, Proceedings of the 6th International Workshop on Aspect-Oriented Modelling, Chicago, USA, March 2005
18. ATLAS Transformation Language (ATL), http://www.eclipse.org/m2m/atl/
19. Objecteering Software homepage, August 2007, http://www.objecteering.com/
20. C. Bockisch, M. Haupt, M. Mezini, and R. Mitschke. Envelope-Based Weaving for Faster Aspect Compilers. In *Net.ObjectDays*, 2005.
21. E. Hilsdale, J. Hugunin, Advice Weaving in AspectJ, Mar 2004, AOSD04
22. VIDE Consortium, *Deliverable number D.1.1: Standards, Technological and Research-Base for the VIDE Project, Project Evaluation Criteria and User Requirements Definition*, 2007

Disclaimer of SAP AG[3]

[3] Applies to Section 3