SPECIFIC TARGETED RESEARCH PROJECT
INFORMATION SOCIETY TECHNOLOGIES

FP6-IST-2005-033606

VIsualize all moDel drivEn programming
VIDE

WP 1	Deliverable number D.1.1 Standards, Technological and Research-Base for the VIDE Project, Project Evaluation Criteria and User Requirements Definition

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 for Information Technology
Leading partner:	SAP AG
Due date of deliverable:	14.01.2007
Actual submission date	
Status	final
Document type:	Report
Document acronym:	DEL
Editor(s)	Andreas Roth, Axel Spriestersbach, Boris Gruschko
Reviewer(s)	Simon Crowle, Sherry Jeary
Accepting	Kazimierz Subieta
Location	www.vide-ist.eu
Version	1
Dissemination level	**PU**/PP/RE/CO

Project supported by the European Commission within Sixth Framework Programme

Abstract:

The VIDE project aims at a visual, Unified Modeling Language (UML) compliant action language, the VIDE language, suited to business applications. This language aims to embed itself in the Model Driven Architecture of the OMG and to be accompanied with a powerful toolset. VIDE is intended to support a business oriented computation independent layer, aspect-oriented facilities, and means for quality assurance.
This first deliverable of VIDE describes the state of the art in all relevant areas for the planned research. It evaluates and selects existing results of state-of-the-art to be (re-)used during further execution of the project. State-of-the-art artefacts which are investigated comprise academic research, standards, and tools. We describe typical user groups, application scenarios, and use cases, and consider especially small and medium-sized enterprises. The state-of-the-art in other core academic research areas, such as meta-modelling, visualisation in modelling, aspect oriented programming, quality assurance, and precise means for describing language semantics, is investigated.
A further focus of the report is the evaluation of the technological basis of VIDE, which is aimed to be based on standards. We thus evaluate the behavioural part of the UML standard, other (de-facto) standards in model-driven software development, such as for meta-modelling, model transformations, and querying models. We also give an overview on tools available for model driven software development. As a result of the evaluation and selection in earlier chapters, we then draw conclusions on the architecture of the VIDE tool.

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 level, service oriented architecture (especially Web services and integration) and business process modelling."* [2]. 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 aims primarily at the domain of business application software.

Work Package 1 provides research on industrial and academic standards as well as software tools and environments for model driven software engineering, such as the Meta-Object Facility (MOF), UML, etc., that the project will be based upon. It introduces possible architectures for implementing the VIDE tool environment based on the Eclipse platforms. In addition, it develops several requirements which are relevant for future work packages – these are summarised in the Appendix.

In this work package, the consortium has investigated the typical user groups in a model driven software engineering approach for business applications and their need with respect to behavioural modelling. We have identified and defined (according to their skills and their expectations towards VIDE) the following user roles: domain users, business analysts, analysts/designers, analysts/VIDE programmers, and architects. For these user groups we have defined typical use cases, ranging from creating a new business application, to modifying or modernising existing applications, and from the implementation of a system to model simulation. Orthogonal to these use cases, we have gathered two typical application scenarios: a sales scenario and a warehouse administration scenario. In addition, the research for gathering small and medium enterprise (SME) requirements (one of the targeted user groups) is planned and will be executed in the forthcoming work packages.

We have investigated the state of the art in other core academic research areas, such as meta-modelling, visualisation in modelling, aspect oriented programming, quality assurance, and the precise means for describing language semantics. Decisions on the selection of specific standards, tools and techniques are deferred to the respective work packages as planned.

VIDE aims at strong compliance to existing standards. First and foremost, this means compliance to the UML standard and in particular its behavioural capabilities, and related standards, such as the Object Constraint Language (OCL). We have investigated these standards and their capabilities with respect to the needs of VIDE, which are in particular the orientation towards the skills of business people and the need to process high data volumes (and modelling the means to deal with them). Also relevant are meta-modelling standards - we have concluded that it is most suitable to use the Eclipse Modelling Framework (EMF) as VIDE's meta-modelling framework. Additionally, VIDE will adopt further meta-modelling techniques around EMF, such as the Atlas Transformation Language (ATL), the XPAND template language, and the Graphical Modelling Framework (GMF). We have conducted an extensive tool survey on MDA tools which has informed precise requirements for tool selection. Finally, based on the investigations, we have elaborated the VIDE architecture from a meta-modelling perspective as well as from the point of view of a VIDE user.

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

Table of Contents

List of Figures

List of Tables

1 Introduction and Overview

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."*[2]. Therefore VIDE investigates *"visual user interfaces, executable model programming, action- and query-language-semantics, AOP and quality assurance on the platform-independent level, service oriented architecture (especially Web services and integration) and business process modelling."* [2]. 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 aims primarily at the domain of business application software.

Work Package 1 provides research on industrial and academic standards as well as software tools and environments for model driven software engineering, such as MOF, UML, etc., that the project be based upon. It introduces possible architectures for implementing the VIDE tool environment based on the Eclipse platforms.
The work package investigates typical user groups in a model driven software engineering approach and their need with respect to behavioural modelling. In addition the research for gathering SME requirements (the targeted user group) is planned and will be executed in the forthcoming work packages.
The results of the research performed is summarised in this deliverable. To guide and synchronize subsequent work it defines a couple of requirements to be fulfilled. The requirements from Work Package 1 are the basis for further investigations and development in the subsequent work packages (2, 3, 4, 5, and 7) that start in parallel.

We provide, after having given an overview on our methodology in *Chapter 2*, a definition of VIDE's target user groups in *Chapter 3*.
A special user group of VIDE are small and medium-sized enterprises (SMEs). How VIDE is approaching their special needs is described in *Chapter 4*.
These users will interact with the system performing certain use cases. Moreover they will act within of certain application scenarios, of which we present two typical ones. Both use cases and application scenarios will be presented in *Chapter 5*.
In *Chapter 6*, we have a closer look at the state of the art in other core academic research areas. We give an overview on meta-modelling, on visualisation in modelling, on aspect oriented programming, on quality assurance, and precise means for describing language semantics. VIDE aims at strong compliance to existing standards. First and foremost, this means compliance to the UML standard, and in particular its action part. This is what is covered in *Chapter 7*, followed by a discussion on other (de-facto) standards in model-driven software development, such as for meta-modelling, model transformations, and querying models.
In *Chapter 8* we give an overview on tools available for model driven software development and in *Chapter* 9 we draw conclusions on the architecture of the VIDE tool, which are consequences of the evaluation and selection in previous chapters.

[2] From the VIDE project summary in the Technical Annex

2 Description of Overall Approach and Methodology

The overall VIDE objectives are:

Objective 1: Introduce a fully visual programming paradigm for data-intense business applications
Objective 2: Support fully UML-based programming
Objective 3: Implement and disseminate tools supporting OMG's emerging Executable UML technology
Objective 4: Support the application of MDA to business systems
Objective 5: Become a "plugin" to modern MDA tools and technologies

In the course of its development, the project will draw on the expertise and experience of its partners to inform and critically appraise the design and evaluation of the VIDE tool. Each partner will utilize their knowledge of the business application domains and the software engineering process to identify the critical and defining needs of business application project stakeholders. This document sets out the VIDE partner viewpoint on these needs, through the exposition of existing and envisaged properties of the business application domain.

To define the VIDE requirements, we first consider the software product from the perspective of a range of users, their activities and work context. This is followed by an indicative selection of application scenarios and use cases that illustrate potential VIDE development contexts in further detail. A theoretical and technical underpinning is subsequently presented that provides a research foundation for the engineering disciplines and methodologies to be undertaken during the development of VIDE. Further refinement of this basis is provided in a description of the formal standards and languages that will be adopted by the project and its implications for the overall architectural view. Following this, an examination of existing tools that support the development of business applications within this context is provided. Having identified the provision of tool-based assistance for business application designers at the MDA computation independent modelling (CIM) and MDA platform independent modelling (PIM) conceptual level, a critical analysis of contemporary software libraries that will support the implementation VIDE modelling tools is discussed. These findings are then set in the context of the VIDE partner software component inventory and, based on a selection of these technologies, a project development environment is selected.

2.1 Defining Users – a VIDE Perspective

It is widely recognised that understanding who your users are is vitally important in the development of the software product (Preece, Rogers et al. 2002). To actually identify those people that may have requirements for any system (that is, to determine who are the stakeholders) is far from trivial. In fact, stakeholders can be identified as anyone that could be materially affected by the implementation of a new system or outcome (Leffingwell and Widrig 2003). This is a much more general definition than specifying users and reflects the Information Systems community views that stakeholders have a much broader effect on systems development than just users. For the VIDE project, identification of the people that may have requirements is particularly complex, in that the partners all have different world views of the type of people that they believe will use the system - perhaps the most common single mistake in development efforts is to leave an essential person out of the process (Gause and Weinberg 1989). The connection between these stakeholders can be seen in Figure **1**.

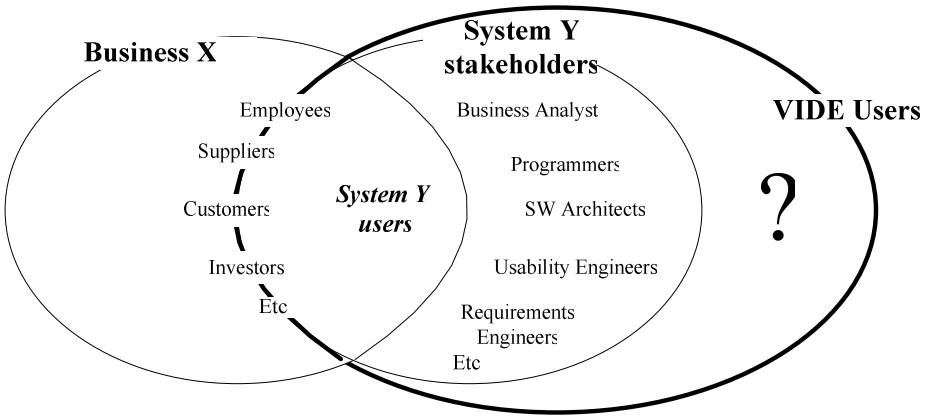


Figure 1: Business and VIDE user roles

Traditionally, the customer and user would be the only people identified as having a requirement on the system. However, it is a mistake to class users as a homogenous group. Two broader groups, containing a selection of roles (as identified by (Avison and Fitzgerald 2006)) are involved in any system development. Firstly, people on the development side, including: programmers, systems analysts, business analysts, project managers, senior IT management and the chief information officer. Secondly, there are those people from a business for whom the system is required. Further definitions in (Avison and Fitzgerald 2006) classify these users as individuals that utilise output or outcomes of an interaction with the system. These will include business users, business management and business strategy management. In addition, there may be external users, who are outside the boundary of the company, which the system will serve. For example, customers or potential customers, information users, trusted external users, shareholders or other sponsors (even the society at large), that are affected by the system.

2.2 Defining VIDE Users

An added complication from our perspective is that since the VIDE toolset may be identifying business processes it would involve a number of intermediaries between a business and the toolset itself. Work carried out by (Avison, A.T.Wood-Harper et al. 1998) when looking at the Mulitview2 framework, suggests that the role of the systems analyst is vital in such scenarios. In this case, the analyst of a system is a role that could be filled by an IS professional, the end-user, or a business consultant. Individuals fulfilling this role are referred to as ‘change agents’ in IS development (Avison and Fitzgerald 2006), and are represented thus:

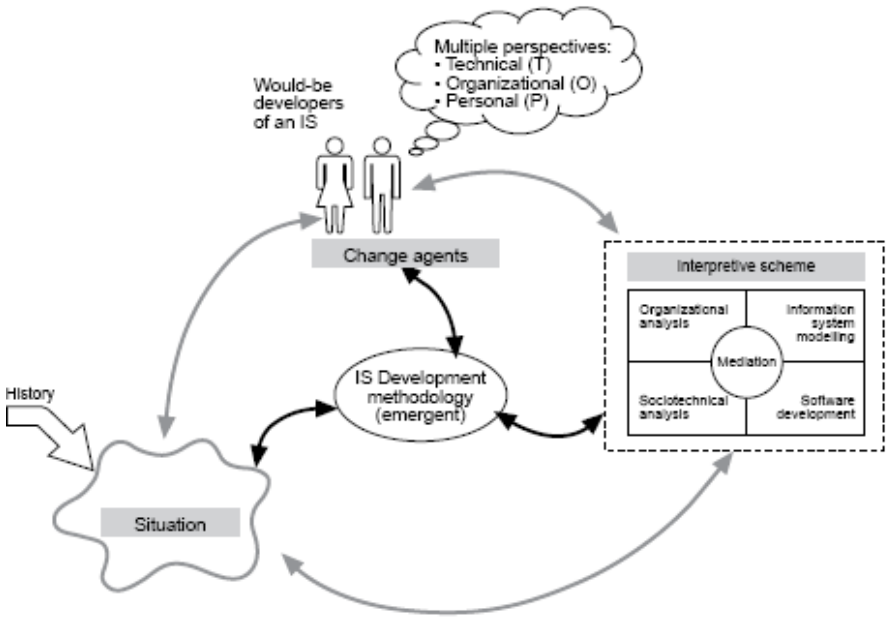


Figure 2: The Multiview2 framework (Avison et.al.1998)

Supporting this role might be seen to be a priority for the VIDE project. Given this, special consideration should be given to addressing the needs of the analyst, who acts as a conduit between developer and client groups. As a liaison between customers and other stakeholders, analysts produce an elaborate documentation of needs; VIDE must integrate these with the development issues that pertain to traditional software development where compilation of source code, and the testing of compiled code is done within one integrated development environment (IDE).

A further dimension to be considered is that of the multiplicity of user experience and varying technical knowledge that must be reconciled within a coherent VIDE development process. Other discriminating stakeholder factors include user skill sets and level of domain knowledge and IT aptitudes; this might be particularly relevant for VIDE, where a breadth of experience is likely (see Section 3.1.2). The advantage of adopting a wider perspective, although perhaps a risk too, is that it is more amenable to rigorous, real-world validation. As a consequence, the training time for VIDE users with a target experience and capability could then be defined.

Contemporary elicitation methods lean heavily on empirical investigation in which data describing a user's domain knowledge, IT skill, work environment and context, task execution and roles in business processes (to name just a few), is gathered. In the absence of any existing understanding of the target user base, an empirical approach that employs these techniques can be critical to the success of the project. However, such methods are time consuming and expensive. For more mature projects, in which developers have considerable experience of the domain and its users, it has recently been suggested that a rigorous, 'activity centred' approach to design be adopted (Norman 2005). Here, the development of software that *clearly communicates its functional role for the essential core activities that affect the successful outcome of domain goals* should be the main focus for developers. More particularly, it may be considered harmful to the overall effectiveness of the software solution if too much emphasis is placed on the demands of a particular user group whose expectations of the tool are not coherent with the actual activity objectives of the application.

In pursuit of capturing the core activities that the VIDE tool will support, we have adopted an approach that frames the partner's knowledge of the expected activities of specific users who would use the tool to support their tasks of developing (often) data rich business applications.

3 Users

3.1 Stakeholders and Requirements Engineering (BU)

Much of the software engineering literature recognises that the requirements phase is particularly important to a successful project outcome and that inadequate determination of requirements is the biggest single factor in project failure. For example, (Glass 1998) notes that many software projects fail due to poor or non-existent requirements processes, and (Hall, S. Beecham et al. 2002) report that 48% of development problems are in the requirements phase. Research in the area agrees that one way to improve the requirements process is to increase the involvement of stakeholders (Nuseibeh, J. Kramer et al. 2003) in both the elicitation of requirements and the validation of specifications.

Indeed, the software engineering community has long understood the importance of stakeholder involvement in validation of requirements and specifications (see (Sutcliffe and Maiden 1993; Leonhardt 1995; Pfleeger 2005)), and some enlightened authors go so far as to explicitly define requirements as "the effects that stakeholders wish to be brought about in the problem domain" (Jackson 1995; Bray 2002).

There is significant work in requirements engineering (and related fields), over many years which supports this view. For example, the CORE approach (Easterbrook 1991) to requirements engineering specifically attempted to make explicit the multiple perspectives of differing individuals (stakeholders). Similarly, Checkland's Soft Systems approach (Checkland 1999) (and its extension by (Avison and A.T.Wood-Harper 1990)), has at is core the idea that different stakeholders will have contrasting worldviews and perspective. In recent years such arguments have been a major factor in the development of scenario-based approaches to requirements engineering. That is, the importance of stakeholder involvement has shifted the emphasis away from their capacity for formal verification towards the comprehensibility of the notation (Phalp and Cox 2002). (Phalp and Cox 2002).

3.1.1 User requirements analysis

Hence, in addressing the classes of business users (user requirements) to whom the VIDE tool must be accessible one must actually consider the breadth of stakeholders who would be involved. Gathering user-oriented requirements is a potentially time-consuming and intensive process; fifteen methods are identified by (Robertson 2001) that specifically relate to understanding user needs. Of these, half require a substantial investment in time directly engaging end users. Since the VIDE project can benefit from the significant expertise and experience of its partners' knowledge of business applications development, three of the approaches described in (Robertson 2001) have been identified as most likely to yield productive results within the constraints of the project.

These approaches are:
Use case descriptions

The case studies presented in Chapter 5 of this document provide illustrations of what are referred to as *business* and *product* use cases (Robertson 2001). The former asserts a basis for

the description of system behaviour based on business related activities. Exemplar cases have been provided to illustrate conceptual components that will be required to be expressed by end users of VIDE in their description of the business problem domain. In the product use case examples, such descriptions refer to the interactions with the solution software itself. Here, the requirements of the VIDE toolset will be informed by the anticipated modelling activities of its users.

Requirements re-use

Use cases and scenarios are a rich and versatile source of information that can inform stakeholders with different perspectives of the impact of user requirements on the system design (Go and Carroll 2004). The analysis of scenarios and use cases have shown to be productive across the software engineering process, including i) product strategic planning, ii) interaction design, iii) requirements engineering and iv) OOA/OOD (Robertson 2001).
By examining historical or indicative examples of business simulations (see Section 5.1) the design of the VIDE product can be further informed. Specific examples of such simulations also engender the exploration of how VIDE users could define CIM and PIM models. The specification of use cases based on the activity driven simulation of typical VIDE use may then generate additional, model-based requirements for the technical underpinning of the VIDE software architecture (see Chapter 9).

Viewpoints

In gathering requirements, it is inevitable that stakeholders of the project will express different points of view. The VIDE partner collaboration is a composition of distinctive perspectives that are inclusive of business process modelling; human-computer interaction; software engineering and enterprise system development. Partner views on the VIDE user requirements are therefore wide-ranging and have been reflected in the definition of user roles, goals, activities and skills (see Section 3.4).

3.1.2 Non-functional User Requirements

In addition to specifying the domain-based, functional requirements of projected VIDE users, a case for non-functional aspects of the VIDE tool is also considered here. A broad characterisation of the general qualities of typical VIDE stakeholders, working at the levels of CIM, PIM and PSM is presented below. They are described in terms of the levels of system engagement specified by (Faulkner 2000) (i.e. *mandatory* and *discretionary* types and *direct*; *indirect*; *remote*; *support* contact) and technical expertise (*novice*; *intermediate* and *expert*).

	CIM	PIM	PSM
Engagement	Discretionary indirect/remote	Mandatory direct/indirect	Mandatory direct
Examples	Business end-users, business consultants/analysts.	System analysts/architects, usability engineers, VIDE programmers	Software developer/engineer
Expertise	Novice	Novice/Intermediate	Expert
Characteristics	Low levels of experience with computers; but high business domain expertise that is only	Low to medium technical experience but have expertise in mapping business or user requirements to	High levels of technical expertise but with typically no *direct* experience of (or

	easily expressed in terms of business goals rather than system requirements.	high-level, system oriented models.	access to) the problem domain.

Table 1 Stakeholders on modelling levels

Novice system users require robust systems that clearly represent their work domain, using visual cues and help to allow them to successfully perform their activities through interaction with the computer (Faulkner 2000). Indeed, VIDE stakeholders contributing at the CIM level are unlikely to be able to express their needs in a technical sense and may not even directly engage with the tool but instead enlist the help of a proxy (perhaps another CIM or PIM stakeholder).

3.1.2.1 REQ – Accessibility at the CIM Level

REQ – NonFunc 1	Accessibility at the CIM level	**SHOULD**
The VIDE environment should provide non-technical, business domain descriptions.		
Description: Non-technical users working at the CIM level should be able to input, retrieve and understand their business domain descriptions in a notation that is non-technical and accessible.		

3.1.2.2 REQ – CIM level Collaboration

REQ – NonFunc 2	CIM level collaboration	**MAY**
The VIDE environment MAY offer collaboration mechanisms.		
Description: It may be possible for CIM or PIM users to collaboratively work on a shared CIM view through a communication mechanism (such as shared notes or links to shared views between stakeholders).		

Those individuals who engage in PIM-level modelling activities are likely to be heterogeneous both in terms of their engagement with the VIDE tool and also their technical expertise. As ‘bridge builders’ between the disparate views of CIM and PSM, this group may contain stakeholders who are predisposed to either the CIM or the PSM perspective but are capable of expressing these concepts to some degree at the PIM level. For this reason, users at this level are likely to require support in their transformation activities, in particular direct access to on-line help and a clear view of the broader aspects of the VIDE architecture.

3.1.2.3 REQ – On-line Support for at least CIM/PIM Users

REQ – NonFunc 3	On-line support for CIM/PIM users	**SHOULD**
Requirement predicate		
Description: Users working at the CIM/PIM level should have immediate access to on-line/in-system, context sensitive help that describes how transformations between CIM, PIM and PSM levels are specified and used in the modelling activities supported by VIDE. Help should be expressed in non-technical terms wherever possible.		

Lastly, it is anticipated that PSM contributors are likely to have high levels of technical expertise and harbour expectations of the system that matches their own experience of system software construction. As a consequence, these users will require visual representations of their modelling and development activities to have significant parallels with contemporary

notations, such as the UML (see requirement REQ – User 2: Reuse of existing UML Standards, Section 3.5).

Irrespective of the modelling abstraction *level* at which activity is undertaken by VIDE stakeholders, the importance of the clarity of its *purpose* and *role* within the overall VIDE business application development process must not be overlooked. Clearly delineating the artefacts of a domain *analysis* and its subsequent interpretation and transformation into a distinct *solution design* model is not well practised within the industry (Génova, Valiente et al. 2005). This criticism of existing practice cites the misuse of use cases, PIMs and PSMs where aspects of the problem domain analysis are frequently confused with the design of the system solution. Specifically, the assumption that the symbolic notation (for example, a UML class diagram) used by one stakeholder cohort will invoke the same interpretation of meaning by another group is regarded as a serious threat to project success (Génova, Valiente et al. 2005).

3.1.2.4 REQ – Clear and Unambiguous Notation

REQ – NonFunc/Semantics 4	Clear and unambiguous notation	**SHOULD**
VIDE should have has clear, comprehensible and unambiguous semantic description suited to the users of the VIDE tools		
Description: The VIDE environment should use notation that has clear, comprehensible and unambiguous semantics suited for the user working at the CIM, PIM or PSM level. Therefore, VIDE must offer model views to the user that do not confound the concerns of one level with another (for example, CIM business process description with a PSM sequence model).		

Appropriately crafted visual representations for each level of abstraction therefore is an important, non-functional requirement for the VIDE project. In order for the system to offer clear representations that are also communicative between project stakeholders, base-line agreements on communicative dimensions (Hundausen 2001) such as *saliency*, *typeset fidelity*, *story content*, *modifiability*, *controllability* and '*referencability*' should be set. These dimensions suggest certain non-functional requirements for VIDE at CIM, PIM and PSM levels. For example, identifying and only representing salient aspects of the business problem at the CIM level (rather than cluttering the user interface with PIM and PSM notation) will support novice users during their interaction with the VIDE toolset.

3.1.2.5 REQ – Model View Saliency

REQ – NonFunc 5	Model view saliency	**SHOULD**
VIDE models views must be user-oriented.		
Description: Views on CIM, PIM and PSM must be controllable depending on specific user interactions with the VIDE environment. It should be possible for users to dynamically control the scope and technical content of these views depending on their specification/comprehension needs. For example, a user working on transformations between CIM and PIM models should be able to work within a view that includes both levels; other, non-technical users should be able to hide such views, or reduce their domain-specific or technical content appropriately.		

The fidelity of the typeset may have an impact depending on the formality of the model being presented by the system; a high quality, text-based presentation will be expected by expert PSM developers whilst the presentation of informal, hand-crafted artefacts is more likely to

be compatible at the CIM level. For similar reasons, story or metaphor-based descriptions of the problem space have been shown to stimulate the discussion and development of conceptual issues (Hundausen 2001).

3.1.2.6 REQ – Appropriate Textual/Graphical Fidelity

REQ – NonFunc 6	Appropriate textual/graphical fidelity	**SHOULD**
VIDE must provide appropriate textual and graphical modalities for its users.		
Description: Users should be able to work with textual or graphical notations that offer the most effective expressiveness for CIM, PIM and PSM concerns. For example, VIDE programmers should expect to be provided with at least a textual editing system that conforms to their previous experience with software IDEs; CIM level users should expect to be able to use a variety of non-technical, possibly graphically rich imagery to describe their business knowledge.		

Modifiability, and its impact on a user's timely interpretation and evaluation of their work (see (Green and Petre 1996) for the original proposition of the cognitive dimensions framework) will have an impact at all VIDE levels; significant changes of one aspect of the solution specification must be communicated to others for the purposes of debugging and traceability.

3.1.2.7 REQ – Timely Feedback and Constraints

REQ – NonFunc 7	Timely feedback and constraints	**SHOULD**
The VIDE environment should provide feedback on user actions at all modelling levels.		
Description: Multiple users working on the same VIDE project should receive rapid feedback on their attempted actions within the VIDE environment. Such feedback should indicate their success or failure to complete an action or task; its impact on their local modelling level; its potential impact on other modelling levels; and any constraints that may impact on the success of their intended action. If this 'ideal' requirement turns out to be unrealistic as a whole in the course of the VIDE tool development, it can be partially skipped.		

During the course of validating the behaviour of the business application, as specified by the VIDE toolset, stakeholders may expect to have high levels of control over the executable parts of the model – simulations of processes or system state transitions should therefore be repeatable or time-shifted.

3.1.2.8 REQ – Runnable and Testable VIDE Prototypes

REQ – NonFunc 8	Runnable and testable VIDE prototypes	**SHOULD**
The VIDE environment should allow execution of runnable models.		
Description: VIDE users should be able to validate at any time (where possible) the models that can be automatically transformed into an executable form. Where possible, executions should be controllable such that users can inspect the properties or states of their model on a step-by-step basis.		

Finally, it must be possible to easily recognise the identity or properties of an entity, and specify its relationship to others within the system (by providing high levels of

'referencability') through simple gesture. For CIM stakeholders in particular, who may have low computer literacy, the ability to express relationships through pointing will enhance their overall interactive experience.

3.2 VIDE User Roles

This section details typical user roles for the VIDE tools. As described in the section above the different user roles may work only on a specific level of abstraction with the VIDE tools. The user roles are based on roles in an MDA process defined by the OMG (Mellor and Watson 2006) and applied to the domain of business software.
A distribution of user roles to the various levels of model abstraction and the main stakeholder addressed by the VIDE project is shown in the figure below.

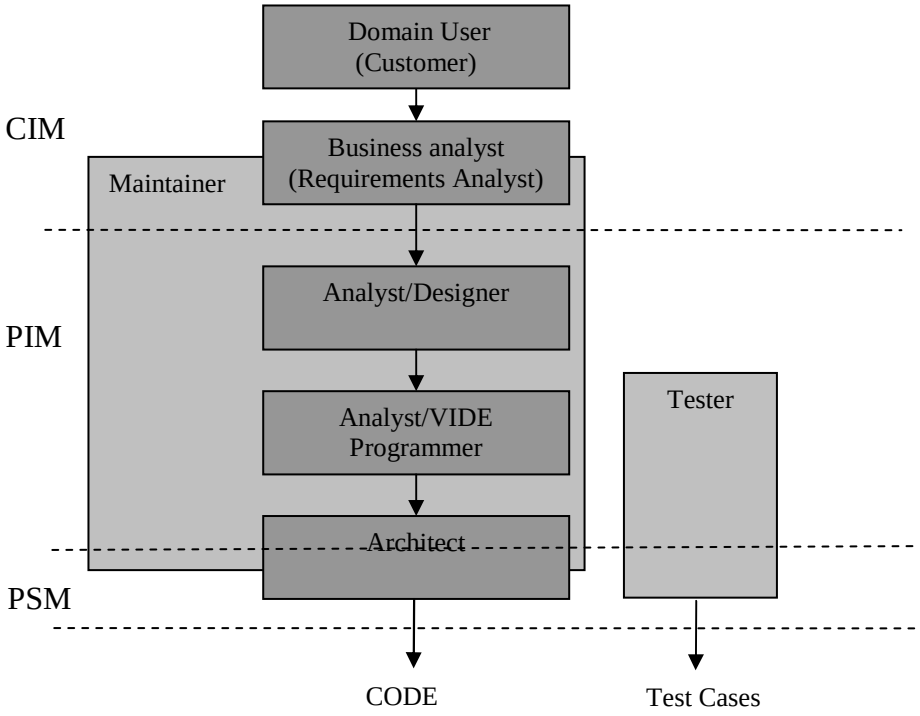


Figure 2 User Roles

The figure depicts the transitional model based development process. Starting from (informal) requirements from domain users the behavioural model is extended in each step and by the specific role until it can be transformed into coding. The user roles are described in more detail in the next sections. The roles "Tester" and "Maintainer" are less important for the VIDE project which focuses on the creation of new software and are therefore omitted from the detailed roles definition.

3.2.1 Domain User (Customer)

3.2.1.1 Description

The domain user is the end user of the constructed software solution. He works for the customer and is an expert in his special domain. For example, an insurance salesman knows about his company's offers and legal regulations and is supported by software solutions without knowledge of technical realisation.

3.2.1.2 Goals

The domain user normally has no knowledge about business modelling but he can draft the requirements for a software application that should be realised with the VIDE tool. In combination with a business consultant the CIM level models can be constructed. The language and the graphical representation should be easy to understand so domain users can validate the correctness of the models. Since domain users typically use specific vocabulary, all tools should support translations into the domain specific language. The domain user serves as a software tester for acceptance tests, i.e. reviews whether a simulated model will perform the expected tasks.

3.2.1.3 Skills

A domain user has special skills in his field of work. He will know about business economics and enterprise management but has normally only office application skills. Experience in modelling of business processes can not be assumed.

3.2.2 Business Analyst

3.2.2.1 Description

Business Analysts advise enterprises on analysis, conception and implementation of IT solutions. They constitute the connection between the customer and the associated IT specialist and need technical as well as social competences. Possible fields of applications are:

- Acquisition and realisation of IT consultancy projects
- Technical and economic evaluation of IT systems
- Change management
- Planning and monitoring of client specific solutions
- Organization of efficient process flow
- Assembly of project teams, leadership and motivation, conflict management
- Planning of manpower requirements and qualifications

3.2.2.2 Goals

A business analyst is one of the main user types of the VIDE tool. They accomplish interviews with domain users and analyse and model the proposed solution on the CIM level. Since they have knowledge in modelling of business processes as well as technical architectures it should be easy for them to use the tool.

3.2.2.3 Skills

Business analysts should have a variety of different skills to fulfil their diverse tasks. They should have knowledge about business processes, modelling and controlling, IT concepts and technologies, procedural models, project management and business economics. Beyond these technical skills, social competences such as leadership, team organisation, partner management or knowledge in legal regulations are required. Typical tools used by the business analyst are business rule management systems and business process modelling tools.

3.2.3 Analyst / Designer

3.2.3.1 Description

Analysts/Designers are responsible for the conceptual model of business entities and the high level business logic. They use the design artefacts and models produced by the business analyst and transform them into a design. The software designer is also responsible for deciding if predefined components may be reused/composed or if they need to be re-implemented. The roles of the software designer and VIDE developer are often combined especially in smaller development projects and organisations.

3.2.3.2 Goals

The software designer is a PIM level expert with a strong background in conceptual modelling and UML class diagrams. The software designer uses the VIDE language and tools to define the first level of behaviour, but leaves the details to the VIDE developer. For reusing or composing new applications from pre-existing components the designer uses the VIDE language (preferring a graphical representation) to understand the business logic that is implemented by a component or to define how multiple components may be composed. Generally the software designers prefer the graphical modelling tools of UML and VIDE for the conceptual and behavioural model.

3.2.3.3 Skills

The software designer is a PIM level expert with strong background in conceptual modelling (i.e. UML class diagrams) and VIDE. Understanding of CIM level artefacts, i.e. the business process model, is also required. Typical tools for a software designer are graphical modelling tools

3.2.4 Analyst/VIDE Programmer

3.2.4.1 Description

The Analyst/VIDE Programmer is responsible for the completing the behavioural modelling that will allow model simulation (i.e. for testing) and the transformation of the models into code.

3.2.4.2 Goals

The Analyst/VIDE Programmer is a PIM level expert with a strong background in behavioural modelling. The Analyst is one of the main users of the VIDE language and tools for detailed behavioural modelling. The Analyst uses the format that is most appropriate for that task. Therefore, the Analyst will make use of the textual VIDE languages when it is more efficient than the graphical representation. Analysts/programmers will also implement components defined by software designers. Therefore they will model the behaviour/business logic of the interfaces that have been designed and also provide the documentation for the components. Analysts will use the graphical and textual VIDE tools.

3.2.4.3 Skills

The VIDE developer is also a PIM level expert with a strong background in modelling especially behavioural models using the VIDE language and its graphical tools.

3.2.5 Architect

3.2.5.1 Description

The Architect is responsible for building the transformations of the behavioural models described using VIDE and the conceptual models into platform specific codings. The architect is an expert in the target platform (for example, Struts) and the programming language (for example, Java) but also has a good understanding of UML and VIDE to be able to define the transformation. An Architect works in application or system development.

3.2.5.2 Goals

The Architect is the expert for the PSM level and has a good understanding of VIDE. The Architect needs to understand VIDE to define the transformations. However an Architect typically does not modify VIDE codings.

3.2.5.3 Skills

The Architect should have knowledge of different target platforms and programming languages. Experience in technical system specification and implementation of the proposed solution is mandatory as well as knowledge about programming concepts like software testing methods for quality assurance.

3.3 Functional User Requirements

The VIDE language allows the specification of behaviour at the model level and thus decreases the need to program. However, users of the language and tools (for example bank, insurance, telecom and aerospace customers) often have specific configurations of targeted languages (C#, Java, C++, C, Fortran, etc.), frameworks (J2EE, CORBA, ...) and development tool chains in place that should be enhanced by the VIDE language and tools. Therefore, both the language and its tools need to be open and highly interoperable to allow working within different frameworks and environments.

3.4 REQ – Flexibility and Interoperability of VIDE language and tools

REQ – User 1	Flexibility and interoperability of VIDE language and tools	**SHOULD**
The VIDE language and tools SHOULD have flexibility and be interoperable with existing tools		
Description: Industrial development projects utilize different programming languages, frameworks and development tools. Usually users have a complex tool ecosystem at work. It is unrealistic to make VIDE replace existing tools. VIDE should thus smoothly integrate with other tools, i.e., read data produced by other tools and provide data readable by other tools. The VIDE language and tools SHOULD have flexibility and be interoperable with the programming languages, frameworks and development tools currently used in industrial development projects.		

In addition to the above requirements, end users are very sensitive to UML standards that are available and partially adapted to the current tools landscape. Therefore reusing existing concepts from UML standard (i.e. existing diagrams, etc.) in the VIDE language is required. A corresponding gap analysis is performed in Sect. 7.2.1.8.2.

3.5 REQ – Reuse of Existing UML Standards

REQ – User 2	Reuse of UML Standard	**SHOULD**
The VIDE tools for certain user groups SHOULD be informed by existing tools for the user groups		
Description: End users are very sensitive to using standards. A key aspect is that the VIDE language reuses as much as possible the UML standard.		

We see a particularly important need in the area of the modelling of the logical architectural level. This is necessary because the definition of the different software systems and components should be expressed in a form which is not prematurely dependent on the software implementation commitments (framework, platforms, language, etc.). In order to have a complete and validated logical architecture, the behaviour must be defined as much as possible, and validated as much as possible. At that stage, the VIDE modelling language can provide the most suitable way to cover the above-mentioned needs. Execution of VIDE model will validate the logical architecture, and parts of the results can be directly reused for implementation purposes.

4 Assessment of SME Requirements

The scope of this chapter is to indicate, define and assess requirements of small and medium-sized enterprises (SMEs) for the adoption of MDA frameworks and methodologies in order to leverage their developments and improve their productivity qualitatively and quantitively while decreasing cost and effort. The selection of the requirements aims to set the goals in order for the VIDE project to be applicable to smaller software vendors that have a specific focus and market.

4.1 Model Driven Development in SMEs

We identify two main types of SMEs regarding their developments and products. The differentiation occurs where the SME is either a reseller of larger vendor products or develops its own products. In both cases, MDA should increase their performance. However, it is more likely that in the first case the vendor provides tools and support in order to facilitate and automate the customization of its products. Therefore in the reseller case, the SME has a predefined MDA methodology, supported by some tools, which will facilitate their development methods.

4.2 Defining and Measuring the Requirements for the Adoption of MDA by SMEs

During the research for the identification of the requirements for MDA, we have identified two different types of requirements that indicate the success of the utilization of MDA from SMEs. First, the adoption of MDA requires additional expenses from small companies that can not afford long term investments. Second, a very important issue is the limited range of application that a SME develops or customizes.

Thus the requirements are divided in two groups that are the following:

- *Cost related requirements*: Requirements that define the real cost of the adoption of the MDA by the SMEs
- *Utility related requirements*: Requirements that indicate the utility for the adoption of MDA compared to former development methods.

These requirements are defined in the following sections.

4.2.1 Cost Related Requirements

Vogel and Mattel (2006) have defined the cost of the adoption as a set of quantitative metrics that enable the quantification of the improvement of an SME after the adoption of MDA. According to Vogel and Mattel (2005) the cost is related to a set of factors that enable the measurement of the direct cost as well as the indirect cost. The latter is based on the quality of tools provided and the company personnel’s competences.

Based on this approach, and because of the limited resources of an SME, we identify the following requirements:

- **Direct cost requirements**:
  a. *Effort*: The reduction of the effort needed for the completion of an implementation project is the most important requirement. Effort identifies the success and the improvement that MDA introduces to SMEs.
  b. *Cost*: The cost also includes the daily cost of the personnel (expenses) used for the development of a project. If the MDA adoption requires only highly trained personnel, the cost of the implementation increases.
  c. *Duration*: The flexibility criterium for the SMEs is related to the capability of the SME to complete implementations in short period of time in order to be competitive.
- **Indirect cost requirements related to tool and methodology efficiency**:
  a. *Tools Maturity*: Tools offering a complete solution in all the stages of the development cycle, and facilitate and automate processes, lead to reduction of the cost of the development.
  b. *Learning Curve*: The capability of the tools to offer a user friendly working environment that does not require very special knowledge, and can be used by non senior staff.
  c. *Perceived value of using MDA*. The new capabilities and limitations that the MDA tool introduces compared to other development tools.
- **Indirect cost requirements related the SME's personnel' competencies**:
  a. *Job description*: The type of the personnel required to work in the development team of the SME that has adopted MDA
  b. *Personnel's Experience*: The experience of the employee required for undertaking a role in the development team
  c. *Personnel's Education:* The education required of the employee required for undertaking a role in the development team

Following the metrics of Vogel and Mattel, this research will utilize these metrics qualitatively in order to identify the requirements' importance regarding the cost of MDA adoption for SMEs.

4.2.2 Utility Related Requirements

Apart from requirements related to cost, this research has identified some requirements related to the utility of the MDA for SMEs. Reduction of cost is not a sufficient argument for an SME to change its development methodology. Thus during this research we have identified some very important requirements for the adoption of the MDA.

These requirements, which mainly focus on the quality of the developments of the SMEs, are the following:

- *Adaptation to the core business:* The limited scope of a SME requires that the MDA tools can be very easily adapted to the core business of the SME. Generic solutions may cover a large vector of developments however SMEs are not interested in generic solutions since they have specific targets.
- *Reusability*: The capability of reusing older projects' developments.

- *Portability:* Portability covers the reusability of older project developments to new environment (hardware or software requirements).
- *Modifiability:* The capability to efficiently extend and customize older development adding or altering existing functionality.
- *Maintainability:* The capability to maintain and support developments, as well as assign them to a new development team
- *Interoperability:* The capability to extend the developments in order to offer functionality that enables integration with other applications.
- *Testability:* The capability to perform testing during and after development.

The adoption of MDA aims to improve efficiently the above requirements and thus the quality of the software delivered by the SMEs.

4.3 SMEs Requirements Assessment

After the identification of the type of the SMEs and the requirements, this research will assess the importance of these requirements based on the type of SMEs.
An evaluation matrix will be used in order to present and assess the requirements according to the above categorization. In the table there are two different types of SMEs (based on Section 4.1) that are the following:

1. *SME reseller*: SME that resells and customizes only 3rd party software.
2. *SME vendor*: SME that develops software

Each requirement is assessed based on importance (● : very important, ◉ : important, ○ not important) and also it is ranked based on its importance (1 : essential to 19 : least important).

	SME (reseller)		SME (vendor)	
	Importance	Ranking	Importance	Ranking
Direct cost requirements				
➢ Effort				
➢ Cost				
➢ Duration				
Indirect cost requirements related to tool and methodology efficiency				
➢ Tools Maturity				
➢ Learning Curve				
➢ Perceived value of using MDA				
Indirect cost requirements related the SME's personnel' competencies				
➢ Job description				
➢ Personnel's Experience				
➢ Personnel's Education				
Utility related requirements				
➢ *Adaptation to the core business*				

➢ *Reusability*				
➢ *Portability*				
➢ *Modifiability*				
➢ *Maintainability*				
➢ *Interoperability*				
➢ *Testability*				

Table 2: Requirements assessment table

4.4 Conclusions

The assessment of the requirements will be done in a later stage of the VIDE project, with SMEs completing the assessment table. The results, apart from identifying the most important requirements for SMEs, will also help to evaluate the VIDE tool and how it meets such requirements, during the evaluation phase of the VIDE project.

5 Application Scenarios and Use Cases

This section describes the usage scenarios of some anticipated applications of VIDE to the development of data intense business applications. First, we describe two typical scenarios for business applications. They serve as a common basis for discussing detailed Use Cases for behavioural modelling and the VIDE language and tools in the later sections.

5.1 Business Application Scenarios

5.1.1 Sales Management Scenario

Sales management systems are used in business to manage all kinds of sales activities in order to increase productivity. Considering this rather huge domain of applications quickly points to many different issues that could be treated in such an example. Examples for only some of these issues are Product Life Cycle Management (PLM), Supply Chain Management (SCM) or Supplier Relationship Management (SRM).
Thus we focus on one specific part of sales management, the Customer Relationship Management (CRM) and even more on specific CRM parts relevant and meaningful for behavioural modelling. The example described is based on examples found in (Buck-Emden and Zencke 2004) and within this document called *Sales Scenario*.

The core concept is the acquisition and exploitation of business process data (i.e., the holistic management). It is irrelevant for our purposes, whether data is only stored and the system provides only a user interface for I/O, or if other communication channels are used to communicate directly or indirectly with other parties involved in a business process (printing invoices, quotations, etc., is considered indirect communication in this regard).
The Sales Scenario focuses on sales processes of enterprises selling one or more products. It involves different aspects, ranging from opportunity management to quotations to customers, sales order processing and invoice processing.

From a customers point of view the functionality of the Sales Scenario is described as follows:

1. A field representative of a manufacturer of computer hardware receives a message on his PDA, telling him that ***company*** *X* is planning to replace its complete system in the next quarter. The company has budgeted substantial financial resources for the replacement.
2. He immediately enters this information in the system, i.e., master data of the potential customer, including budget estimation, description of sales opportunity, sales volume, and timeframe of the opportunity.
3. The ***Opportunity*** object is created in the system and evolved by the assigned employee until it reaches a go/no go decision by sales management.
4. Another employee of the sales office creates an offer using the ***Quotation*** module, which automatically generates a quotation template based on the sales opportunity.
5. Based on the categorisation of the prospect in a Customer Group, estimated sales volume, and sales probability, the ***Individual Prices*** module is used to calculate a discount for the customer, which is included in the quotation.

6. After the sales office has contacted the customer and received an order, the system automatically converts the quotation into an order upon mouse click using the module ***Sales Processing***.
7. To check the creditworthiness of the customer, a ***Credit Check*** is performed during sales processing by interacting with the ***Payment*** module.
8. An (optional) ***Availability Check*** is performed to check warehouse stock for required capacities.
9. The availability-to-promise check requires interaction with warehouse management (***Stock***). In case of ***Multiple Stocks*** only those warehouses sufficiently close to the shipping address are included.
10. In cooperation with the ***Delivery*** module, the order is split into separate orders for each involved warehouse, which have to be scheduled appropriately.
11. If ***Payment*** is to be integrated into the process, it would be activated automatically upon creating a binding sales order. Depending on the method of payment offered by the system and selected by the customer, an automatic debit transfer from the customer's account can be triggered (***Payment Card***), an invoicing document can be attached to the delivery (***Cash On Delivery***), or ***Invoicing*** is activated for later settlement.
12. The order status is set to "completed" by an employee as soon as it is delivered to the customer.

All the described process steps which involve concrete documents are influenced by available communication channels. This determines, e.g., whether an invoice is created separately using Word, printed from the system, or automatically sent via e-mail. The following business components could be tackled in more detail:

- Opportunity: This Opportunity module manages the evolution from an initial customer contact (in the example for replacing the computer system) towards a decision to send a Quotation to the customer. This includes various steps of customer engagement (e.g. mail, appointments…) that should evolve Opportunity so a decision by sales management is possible. Typically behaviour for an Opportunity is for example the evolution of the opportunity state (active, stalled, failed) and calculation on the predicted value.
- Payment. This module manages the payment process, both for incoming and outgoing payments. It manages for example receipts, remainder calculation. It is possibly integrated with the banking system for instance to trigger money transfers and interacts with invoicing.
- Customer Group. Customer groups influence Pricing and offered product portfolio (Product Management). Competitors, for instance, are either not offered certain articles at all or at special rates only.
- Product Management. This includes the management of the product portfolio, possibly as one or more product lines. Products are registered in the system and augmented with meta-data such as name, identifiers, units, category, manufacturer, taxes, certificates, minimum, maximum and actual "on stock" values, pricing, alternative products and more. The management of aggregate products is possible with or without warehouse management, i.e., stock of certain products is composed of stock of

included products (e.g., product bundles as special offers or even real complex products consisting of parts, cf. computer hardware shop)

- Stock. A stock manages product inventories in one or several warehouses. It has a direct relation to inventory/capacity of products described in Product Management. It contains additional information on storage location (aisle, shelf, etc.) of individual products.
- Customer Order Management manages processing of sales orders from customers, it thus includes accepting orders, shipping of sales.

Both behaviour modelling and the VIDE language will be used described the business logic/business behaviour implemented by the systems. For example, in Invoice Processing the system must guarantee that the invoice will not be issued to a customer unless all products with the required number are in stock. Action semantics enables the specification of those actions on PIM level. However some issues (e.g. data be stored persistently or should be executed within a transactional framing) require specifications in the platform-independent VIDE language accessible for users with little or no IT background.

5.1.2 Warehouse Administration Scenario

5.1.2.1 Summary

This scenario covers a simple warehouse administration tool and the business process that should be supported by this tool.

5.1.2.2 Detailed Description

In this use case the data intensive administration of a warehouse should be supported by application software. The software should be used for several warehouses. Therefore different storage capacities have to be taken into account by the software. The following figure shows the event driven process chain (EPC) of the operation at the CIM level.
After a delivery has arrived its invoice is checked. If the invoice is not correct the delivery is sent back. In any case the result of this check is stored. After storing the result, the master data for each good of the delivery are checked concerning availability and correctness. If necessary these data are created or corrected. Then the goods are put in the warehouse storage and added to the list of stored goods of the according storage space.

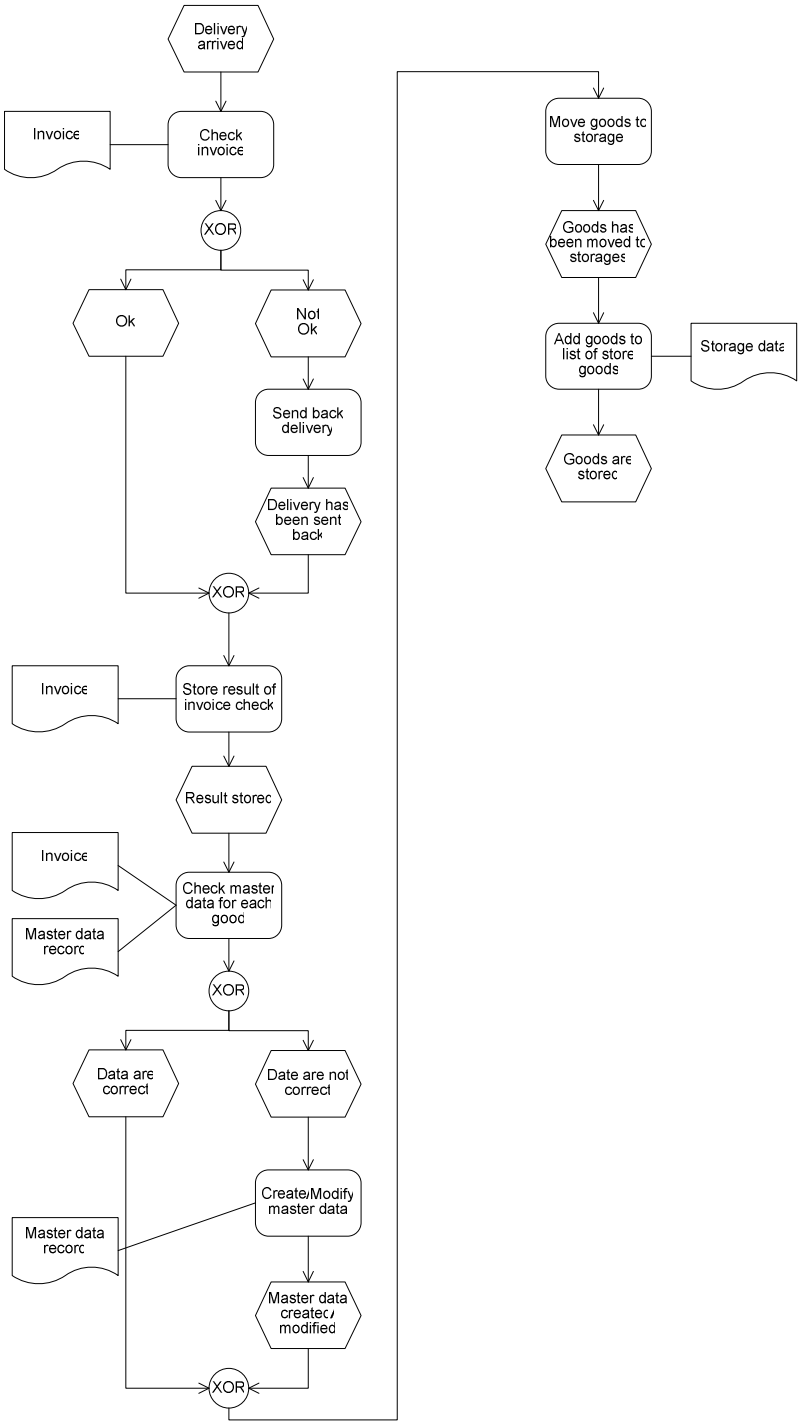


Figure 3: An EPC model describing the warehouse scenario

Additionally to the process description the following requirements have to be fulfilled:

1. A master data record for every good
2. Storage places
3. Invoices can be
    a. created
    b. edited
    c. deleted

The following data are required:

- For each master data record
  - Name of the good
  - Unique identification number
  - Price
  - Weight per unit
  - Packing type (box, pallet, barrel)
  - Due date
  - Frozen (Boolean)
- For each storage
  - Name
  - Coordinates into the warehouse
    - Row and column
  - Packing type the can be stored
  - Capacity
  - List of stores goods
- For each invoice
  - Unique invoice number, which contains the actual year at the beginning
  - Date (formatted DD:MM:YYYY)
  - Address of sender
  - List of goods, with
    - Price and
    - Amount for each good
  - Price total
  - Correct delivery (Boolean)
  - Comments

The scenario can (and is in reality) extended as follows:

- In addition to what is described above, receiving a correct delivery requires notifications to accounting and to processing of payments.
- Moreover storing must be operationalised by concretely starting site logistics processes.
- Finally, the checking of the quality of the delivery can be incorporated.

5.2 Use Cases

This section describes use cases for users using the VIDE tools when implementing the Business Application Scenarios described above. Each use case is described in a separate section. Each of the use cases contains information on

- the target user group in compliance to the user roles identified in Section 3.2,
- a summary of the use case,
- the (detailed) use case description, and
- the required VIDE tool support for the use case and the targeted user group.

5.2.1 Model Simulation

5.2.1.1 Target User Groups

Business analyst
Analyst/Designer
(Analyst/VIDE Programmer)

5.2.1.2 Summary

Execute the model of an application in order to validate its logical behaviour.

5.2.1.3 Description

The VIDE language is used for model simulation and validation, before software implementation takes place. The typical customer will not immediately replace a traditional programming approach by a completely model-driven approach. The main reason is that he needs to trust the generated code, and to feel confident that he masters the development process. Therefore, the start is to model the business level with CIM models, to define a complete behavioural modelling on the PIM level and to validate it using a VIDE code generation in order to simulate its future behaviour. In the logical modelling, technical aspects such as transactional issues, safety and security, data storage optimization, usage of specific platforms, etc., are usually not considered. Only the logic of the system is to be considered. This use case can take any usual business application, such as sales management systems, or delivery management systems, and define its logical model. Put briefly, the scenario is: define the logical business model, then generate the code and simulate that model.

5.2.1.4 Required VIDE Tool Support

The VIDE tool should support modelling the logical business level (CIM), generating models on the PIM level, and simulating the model on that level.

5.2.2 Provide a Final Implementation

5.2.2.1 Target User Group

Analyst/VIDE Programmer
Architect

5.2.2.2 Summary

Use VIDE to provide a final implementation on a real target.

5.2.2.3 Use Case Description

Connect the VIDE model and its generated code from the previous use case to an existing platform (such as J2EE), and implement the application, reusing as much as possible the generated code, adapting the model to obtain a dedicated PSM and having a better suited regenerated code.

5.2.2.4 Required VIDE Tool Support

The VIDE tool provides modelling facilities, integrated in a state-of-the-art IDE, and code generation.

5.2.3 Construction of Business Software Applications

5.2.3.1 Target User Group

Analyst/Designer
Analyst/VIDE Programmer
Architect

5.2.3.2 Summary

Model and implement the scenario in an existing business software architecture

5.2.3.3 Use Case Description

We are considering the following simplified architecture of business software:

- Business objects are real world entities modelled as objects[3]; they have specific types comparable to classes in OO.
- Business objects can be changed by invoking atomic services; services consist of several service operations[4].
- Business objects are hierarchically structured as trees consisting of a root and items.
- Business objects are contained in process components. Process Components group business objects that are semantically related. They represent the software realization of a business process.
- Business objects in different process components communicate via message exchange. In the case that a message must be sent from one component to the next there is a special infrastructure (process agent framework) in each component which decides whether a message should be sent and whether an incoming message should be accepted and processed.

The components which realise the functions required for the application scenario described above follow this architecture. For instance, in the warehouse scenario, “Invoice” is a business object, as well as “IncomingDelivery”. These business objects are contained in different process components and communicate via message exchange.

5.2.3.4 Required VIDE Tool Support

- Support for modelling the described flow from a business perspective in a CIM accessible to business experts (like in Figure 3).
- A model on the CIM level is transformed into a PIM model.

3

http://help.sap.com/saphelp_webas610/helpdata/en/59/ae4484488f11d189490000e829fbbd/frameset.htm

4

http://www.sap.com/platform/netweaver/pdf/WP_Enterprise_Services_Architecture_Intro.pdf

- Given a concrete static structure, which divides the domain into process components and business object, a PIM model is generated from the CIM model. The PIM model is enriched by additional behavioural information.
- Simulation of the behaviour of the process component on the CIM and the PIM level as in the Use Case described in Sect. 5.2.1.
- The enriched PIM model is transformed into a PSM level model which is compliant to the architecture described above (optional)
- generate executable code or code skeletons (optional) as in Sect. 5.2.2.

The use case could as well prove how flexible VIDE is to take care of crosscutting variations. For instance it might be possible to support “a shipping notification”, which adds functionality at several places in the process, such as when the delivery arrives, when it is accepted, etc. Maybe this variation can be modelled with aspects.

Further use cases build on the architecture described above.

5.2.4 Extend an Existing Application

5.2.4.1 Target User Group

Business Analyst
Analyst/Designer
Users of this use case are at the same time adopting the role of a maintainer.

5.2.4.2 Summary

Use VIDE to in conjunction with an existing application which model is known.

5.2.4.3 Use Case Description

Most use cases do not build new systems from scratch, but extend existing applications. We start from an existing application, where the model is known. We extend the model of the application using VIDE. The VIDE model is used to simulate the logical model, with pieces connected to the existing application.

5.2.4.4 Required VIDE Tool Support

VIDE supports the users while modelling, by integrating new models with the system to be extended. Simulation of the system using VIDE generated models and subsequent code generation facilities (that include the extant code) complete this level of support.

5.2.5 Process Extensibility

5.2.5.1 Target User Group

Business Analyst
Analyst/Designer
Users of this use case are at the same time adopting the role of a maintainer.

5.2.5.2 Summary

Referring to the system architecture described in 5.2.3, additional process steps are added on the CIM level and realised with tool support as code.

5.2.5.3 Description

With standard business software, often special adaptations have to be made for specific customer needs, without changing the integrity of the original system. As an example, the warehouse scenario, could be modified for a customer as follows. When a wrong delivery is sent back, a special process component and business object is notified which maintains relations to suppliers and which in particular maintains a repository of problems with a certain supplier. This repository may be a source of potential changes in the policy towards this supplier. Again this new functionality must map to the business software architecture as described in Sect. 5.2.3.

5.2.5.4 Required VIDE Tool Support

The VIDE tool should help the business expert by providing a graphical modelling interface, and by offering a graphical simulation environment. The special issue in this scenario is to bridge the gap between models on the technical domain (platform independent models – PIMs) and models on the business level (computation independent models – CIMs), denoting business processes or workflows. This mapping is a major challenge in this scenario.
The adaptation should as much as possible be conducted on the CIM level. That is, the business expert models the new additional process component, specifies its business objects, models additional process agents of the existing process component, generates adaptations of the models of the existing components. The new component is then modelled with the VIDE action language and relevant models and code are generated.

5.2.6 Modernise Existing Applications

This VIDE use case is optional.

5.2.6.1 Target User Group

Business analyst
Analyst/Designer
Analyst/VIDE Programmer
Architect

5.2.6.2 Summary

Modernize existing applications using VIDE.

5.2.6.3 Use Case Description

In this case, the model of the application is not well known, or not up to date with the application. (Which sadly is by far the most frequent case within organizations).
An existing application can be reversed into UML, in order to redevelop some pieces. One first action might be to ‘componentise’ the reversed model, in order to isolate unchanged parts from regenerated parts. Then, the code is generated for the modernized parts, and connected to the existing application.

5.2.6.4 Required VIDE Tool Support

VIDE integrates into a UML case tool with reverse engineering facilities.

6 Academic Research Base

6.1 Model Driven Software Development

Since its announcement by the OMG in 2001 (Object Management Group 2001), MDA is viewed as an important subject of research (Atkinson and Kühne 2003; Bézivin, Hammoudi et al. 2004; Gasevic, Djuric et al. 2005) (Wegmann and Preiss 2003; Kovse and Härder 2004) as an emerging standard for application development focused on PIMs. The central idea is to utilise software models to drive development. That is, rather than being a passive overhead, models will be the key development tool, offering a high level of abstraction and significant productivity gains. The latter depends on the completeness of PIMs and on the extent to which their transition towards the PSMs of any chosen implementation platform can be performed automatically. These features lead to the postulate of 'executable software models'. Note that similar promises were formulated but not realised back in 1980s, with both executable specification and enactable CASE tools. With today's mature and uniform modelling standards together with the evolutionary rather than "all or nothing" nature of MDA, the main obstacles are removed and a certain level of productivity improvement seems to be guaranteed. The momentum currently observed in the MDA field will make it one of the dominant technologies in the near future. Recent report of Gartner Group positions Model-Driven Architectures among the technologies in the rise phase (Gartner 2006). As with most technologies in the IT-industry, the internal and external quality of the models is of great importance for maintenance and reliability. Successful MDA is expected to make the models the main development artefacts, replacing today's programming languages. This is analogous to the way high level programming languages have previously replaced assembly languages (Mellor, S. J. and K. Scott, et al. 2004).

The way to produce executable PIMs, called Executable UML (Mellor and Balcer 2002), has its roots in the Shlaer-Mellor methodology (Shlaer and Mellor 1991; Mellor and Balcer 2002) and in various approaches from the technical software and real time fields, such as SDL (Berry and Gonthier 1992; Selic, Gullekson et al. 1992; International Communication Union 2002), ROOM (Selic, Gullekson et al. 1992) and the statechart (Harel 1987) approaches. The authors of the Executable UML concepts (Mellor and Balcer 2002), previously applied to the OMG with their proposal for the action semantics (Alcatel, I. Logix et al. 2001) which was finally adopted in UML 1.4 (Object Management Group 2002).

In (Mellor and Balcer 2002) action language code is used to specify application logic. In this pioneering work on executable UML, the authors use action language code solely to specify actions which happen after an object changes its state to a different one. This is a surprising limitation which most be overcome by VIDE, since state machines are sufficient to cover only some part of real-life situations. This first action semantics standard resulting from compromises between different modelling approaches has been integrated into the existing UML framework. Within the UML 2.0 standard (Object Management Group 2004), an effort has been made to better integrate the action semantics, and to provide a global consistency with the dynamic models of UML (state, activity and interaction). Further standardization efforts needed to realise model execution have been initiated (Object Management Group 2005). Efforts are currently underway to provide a support for a high level query language within the OMG, partially based on the OCL standards, both in the MDA field with the Query-View-Transformations (QVT) standard (Object Management Group 2005), and in the

business area fields with business rules related standards. However, there is still the need for higher level query languages, accessible to end users.

The multitude of tasks outlined for model-driven development necessitates the cooperation of a number of specialized MDA-aware tools. This generates the need for precise metadata exchange. The MOF (Object Management Group 2002; Object Management Group 2004) based XMI standard (Object Management Group 2005) addresses this need, although there are currently implementation differences among vendors, which may undermine interoperability. The MODELWARE project (Millot 2004) gives special attention to those issues with its Model Bus architecture (Blanc 2005). In addition to sorting out the XMI compliance for model exchange, an API style communication was proposed for finer granularity metadata movements. Another key research and standardization area of MDA is the development of uniform and efficient means of model querying and transformation (Gardner, Griffin et al. 2003).

Some exploratory research in the area of platform independent AOP and the use of AOP in a MDA has been performed (Eichberg 2002; Kulkarni and Reddy 2003; Wampler 2003; Barbosa, Contreras et al. 2005). The domains of the aspect and the model community overlap (Stein, Hanenberg et al. 2004; Han, Kniesel et al. 2005; Völter 2005). Initial research is on going, but no conclusions on how to best merge AOP and MDA have been drawn. Some sample tool implementations exist which implement AOP in an MDA setting (Barbosa, Contreras et al. 2005). It is likely that the future will merge AOP and MDA into a new paradigm, or will extend the MDA paradigm to crosscutting concern semantics (Mezini and Ostermann 2005). Both are powerful methodologies, which solve existing problems in current mainstream paradigms like OO.

6.2 Human Computer Interaction and Visual Programming Tools

The concept of VIDE as a fully visual language for application building is also underpinned by other research fields, frequently addressed by various researchers in recent years, such as:

- **human-computer interaction** (McCrickard, Czerwinski et al. 2003) (Czerwinski 2002; B. Shneiderman 2003),
- **visual user interfaces** (Braubach, Pokahr et al. 2002; Kang, Plaisant et al. 2003; Marcus, Feng et al. 2003)
- **visual programming** (Gordon, Biddle et al. 2003; Blackwell, Burnett et al. 2004; Guibert, Girard et al. 2004; Ko, Myers et al. 2004; Rosson, Ballin et al. 2004; Carlisle, Wilson et al. 2005; Lawrance, Clarke et al. 2005).

Each of these large, interdisciplinary research domains have the potential to inform the VIDE development process throughout its lifetime. Whilst there are a wide range of human-computer interaction considerations that the VIDE partnership could consider, a primary concern is that of appropriate graphical notations for its end users. Recent research (Hendry 2004) suggests that graphical design representations are very important for ‘boundary objects’ (project artifacts that are discussed by stakeholders with differing domain expertise). Boundary objects are said to be focal points for the clarification of system design issues, particularly during elicitation; problem and solution framing; solution evaluation; and system documentation. Critically, (Hendry 2004) finds that stakeholders do not necessarily use a single, synthetic language but choose to use ‘special-purpose’ (domain oriented) notations to exchange knowledge between discipline boundaries. For VIDE, this indicates that the user

interface must support a rich, and to some degree customizable (or at least include the facility to annotate) visual notation.

As previously discussed (see Section 3.1.2.4) the end-user notation should be communicative and exhibit qualitative properties appropriate to the user and the levels of abstraction they specify within the VIDE framework. For the non-technical VIDE stakeholders, sketching an informal expression of needs is likely to be a modality within which they are most comfortable. Here, visual metaphors have been shown to be effective not only for usability in a broader context (for example see (Ark, Dryer et al. 1998)) but more specifically, metaphor is consistently used as a means of expressing abstract computational processes (Hendry 2006).

Graphical presentations of computable algorithms are considered more formally within the visual programming languages paradigm. This paradigm is an emerging area of research that addresses a significant part of the overall software engineering process. Results have been mixed; in (Catarci, T. 2000) there is a significant criticism about the usefulness of visual interfaces for querying by non-IT professionals. The conclusions of this paper should be considered and thoroughly investigated. It should be stressed, however, that the paper deals with visual query languages and visual interfaces used for specifying queries which display their results immediately after the queries are formulated. VIDE utilizes query languages for a different purpose. There is a lot of research concerning **visual query languages** (Blau, Immelman et al. 2002; Leopold, Heimovics et al. 2002; Smith and King 2002; Stolte, Tang et al. 2002; Abraham 2003; Barclay, Griffiths et al. 2003; Erwin 2003; Kules and Shneiderman 2003; Trzaska and Subieta 2004). Such research is being undertaken also in European projects, such as the ICONS project (IST-2001-32429). In VIDE, we create, edit and document action code for an application and do not deal with an immediate visual representation of query results, but in visual representation of action code. This also means a different target user group – one for whom metaphors may be more appropriate (these have been shown to enhance so called ‘end-user programming’, see (Ko, Myers et al. 2004)).

The term **visual programming** is tightly connected to **end-user programming**. According to (Mayers, Ko et al. 2006) end-user programmers are people who write programs, but not as their primary job function. They develop (small) applications because they want to perform some business tasks (from their primary job point of view) such as accounting, insurance, stock analysis, etc. Usually those people prefer visual languages rather then fully-fledged programming languages like Java, C# or C++. The VIDE language will be an ideal solution for such workers. As stated in (B. G. Ryder 2005) visual programming properties could be achieved by using special environments or by introducing domain-specific visual programming languages. VIDE belongs to both these groups: we are going to develop a special Integrated Development Environment (IDE), which will be syntax-directed for both textual and graphical expressions. In addition there will be the VIDE language itself which is dedicated to defining both business logic and domain-specific applications. Moreover, VIDE could be extended to cover domain specific needs.

Another aspect of VIDE is **code-less programming** (Mezini and Ostermann 2005) enabling the creation of applications without hand coding, i.e. replacing hand coding by some other activities involving more **interactive** communication with the computer. In addition, individual groups have attempted to build enactable models of parts of the UML, e.g., (K.T.Phalp and Cox 2003; Kanyaru and Phalp 2005) but these efforts have tended to focus on the requirements phase, rather than providing support through to development.

The industrial Rapid Application Development (RAD) tool Magic Developer[5] is an example of a programming environment in which code operating on databases is specified in an interactive way without a text-based language. The idea of Magic can be stated as follows: "Generalize the typical code working on a database to a very general loop, which can be parameterized to deliver particular cases of this code". The code in Magic is held and edited in a big number of windows (this makes an overall overview of the code difficult). Editing entries in these windows is performed in an interactive way – this means that the editor always displays a set of choices the developer picks from. This makes the coding easier and reduces the number of errors which can appear during coding (strong type- and context-checking). The VIDE editor will have a similar advantage. Contrary to VIDE, Magic does not have neither a textual nor a visual version of code. Moreover, Magic is neither object-oriented, nor embedded into UML. Nevertheless, the successful use of Magic for rapid development of business critical database applications shows that a fully interactive code editor capable of creating database-intense applications without any need of hand coding is a reality.

Modern user interface design techniques and tools need well defined requirements formalization as the basis for the development of 'model based' architectures for specifying interactive systems. Model-based specification is typically graphical in nature, and addresses stakeholder concerns such as workflow (Stavness and Shneider 2004), task modelling (Luyten, Clerckx et al. 2003) and human-computer dialogues (Traetteberg 2003) (Griffiths, Barclay et al. 2001). As such, contemporary model-based approaches to user interface design seek to integrate usability concerns within the broader framework of MDA; they promote abstraction, automation and platform independence in the design and implementation of user interfaces. Developments in this area indicate the future directions for visual representations of complex, interactive systems. The need to provide usable interfaces is clearly vital, and this requires an understanding of the metaphors employed by such approaches (Crowle 2004).

A lot of research has been done and is under way in the field of human computer interaction, visual querying and visual programming. The VIDE project will be based upon this research. In the current research, however, the fully visual development of the logic of data-intense business applications is missing. The proposed project is intended to fill this gap and contribute to software development standardization efforts.

6.3 Aspect-Oriented Programming and Modelling

6.3.1 Introduction

Within the VIDE project aspect-oriented modularization concepts, suitable for platform independent modelling and composition, will be integrated into a model-driven development tooling. The overall goals are to select the most reasonable aspect-oriented composition mechanisms, define a suitable semantics of aspect modules for platform independent models and identify expressions in UML action languages triggering aspect-oriented adaptations.

In this project we neither develop a new methodology or tool support for identifying, specifying or documenting aspectual requirements, e.g. aspect mining, nor invent a new methodology or process for aspect-oriented (AO) design. In the focus of this project are

[5] http:// www.magicsoftware.com

- visual representation of AO language constructs at design level, i.e. explicit connectors, crosscutting interfaces, join points etc.
- specification of elements of design aspects, such as attributes, methods, advices, pointcuts, inter-type declarations etc.
- selection of join points within structural and behavioural models
- integration of aspect-oriented composition into the model-driven development process

This section describes the research base of Fraunhofer FIRST which will be extended in the VIDE research project. It introduces general terms, core concepts, semantics and mechanisms of aspect-oriented programming at a conceptual level. Furthermore an overview of existing approaches to aspect-oriented modelling and aspect-oriented composition at design level is given.
A more detailed analysis of aspect-oriented modelling will be conducted in the course of Work Package 3.

6.3.2 Core Terms and Concepts

In Aspect Oriented Programming (AOP) a huge variety of techniques and concepts is employed to achieve aspect-oriented modularization. Often similar terms denote different concepts. This section introduces general terms and concepts of aspect-oriented composition at a conceptual level.

6.3.2.1 Core Terms

Four terms form the basis of any description of aspect-oriented modularization concepts. In this section the basic understanding of aspect, join point, pointcut and advice is defined. Most definitions are derived from the definitions in (Klaas van den Berg 2005)

6.3.2.2 Aspect

An aspect is an unit for modularizing an otherwise crosscutting concern (Klaas van den Berg 2005). It defines structural or behavioural enhancements that are attached to another unit. Most often, an aspect module provides new features, such as pointcut and advice, to define those enhancements.
The aspect module may influence the AO composition in three different ways (Schauerhuber, Schwinger et al.). An aspect:

- may act as base themselves,
- might be specialized into several sub-aspects, and
- may introduce adaptations that cause conflict.

6.3.2.3 Join Point

In AOP join points are considered as well-defined "points in the execution of the program" (Xerox) where aspects can interact with other parts of the program. The execution of models requires an adequate definition and considers model elements rather than program elements. Similar to executable program elements, such as statements or expressions, every structural and behaviour model element that appears in the execution of the model can act as a join point. Elements of a structural diagram may represent a join point shadow, specifying where an aspect adaptation can be introduced. A join point shadow with no further restriction acts as join point in every model execution. Model elements of behavioural diagrams directly

represent specifiable join points. They depict the execution of model elements within a certain scenario. Both kinds of elements are used to formulate an AO adaptation.
A join point model defines all elements that can act as join points during model execution.

6.3.2.4 Pointcut

A pointcut is a predicate that matches join points (Klaas van den Berg 2005). Since, join points are points in the execution, they comprise static (structure related) and dynamic (execution related) properties. Two kinds of pointcuts can be distinguished: (i) pointcut that select join points by specifying their static properties, i.e., properties of their join point shadows, and (ii) pointcuts that refer to dynamic (runtime) properties, i.e., properties of an specific join point shadow execution.
A pointcut is often a member of aspect modules.

6.3.2.5 Advice

An advice is an artifact that augments or constraints concerns at join points, (Klaas van den Berg 2005) or IOW an advice is the actual behaviour to execute before, after or around a join point (Filman).
An advice is very much like a method. It defines a list of parameters and contains a block of statements that are executed when the advice is invoked. However, in several AOP approaches advices don't have a name and also no return values. An advice is often a member of aspect modules.

6.3.3 Core Concepts

Aspect-oriented composition is generally achieved by combining two model elements. The resulting model element comprises the structure and behaviour of all the elements with which it was composed. In which way the structure or behaviour of a particular model element is adapted, i.e., augmented, modified or replaced, is specified by the composition. In general, two specific model compositions can be distinguished: merge of different module structures and the adaptation of a module's behaviour.

6.3.3.1 Structural Composition

The structure of the resulting elements is produced by merging the structures of two (equivalent) model elements. This symmetric composition allows the introduction of new members and declaration of new module relationships. In contrast to the programming level, also relationships between model elements can be merged as first-class entities.

6.3.3.2 Behavioral Adaptation

An aspect adapts the behaviour of a model element at a specified join point. This asymmetric composition is specified by a pointcut and binds an advice to a set of join points. The pointcut specifies at which join points the aspect modifies the existing behaviour, and the advice defines the additional behaviour that is executed before, after or around the join point. Behavioural adaptations are in general only navigable from the aspect's side.
In AOP the actual composition is called weaving, which can either by static (at design time) or dynamic (at runtime).

6.3.4 Aspect-oriented Modelling

Currently aspects are used on a programming language level. With the advent of model driven development and the increasing focus on modelling, several research groups tried to move aspects to the model level. With Aspect-Oriented Modelling (AOM) aspects are integrated in model driven development methodologies.

6.3.4.1 Kinds of Models

Several approaches exist to AOM. The first approach is to model a specific programming language aspect framework like AspectJ with UML (Han, Kniesel et al. 2005). This results in AspectJ typical artifacts and thinking. The second approach is to abstract aspect-oriented development and move it to a conceptually higher level (Chavez and Lucena; Clarke and Walker). After this, the aspect model, composition model, advice model, execution semantics and aspect interactions are expressed in a framework independent way and modelled too (Schauerhuber, Schwinger et al.).

6.3.4.2 Notations

Aspects can be modelled with different notations. The most common way is to use a visual notation. This is achieved by extending and customizing UML with UML meta-models and profiles (Fuentes and Sanchez). For easier tool support and better user acceptance for most people this is the preferred approach. If UML and the UML extension mechanisms are not flexible enough, aspects can be modelled visually with a special custom notation. Most current approaches only use class diagrams for AOM modelling (Zhang 2005) and therefore only model structural not behavioural AOP.

6.3.4.3 Modelling Level

As mentioned, aspects are currently used on the programming language level. When pulling them up to a model level, they can be modelled on a computation independent model (CIM), platform independent model (PIM) or on the platform specific level (PSM). Each level has different constraints on the modelling of aspects and needs different artefacts and probably different visual notations. Beside structural AOP modelling (especially the CIM level) needs behavioural AOP modelling, for example AOP annotated use cases.

6.3.4.4 Pointcut Languages

Pointcuts connect join points in the target model with aspects. Those connections are crucial in AOM (Stein, Hanenberg et al. 2004; Rashid, Garcia et al. 2006). On the programming language level pointcuts are described with text for example with regular expressions for matching join points (Jackson and Clarke 2006). Moving to a model level, pointcuts can also be modelled visually. There do exist several visual pointcut languages, which either directly associate join points with aspects or provide a visual querying language for join points (Stein, Hanenberg et al. 2004; Zhang 2005), which then connects the visual query description with aspects. Another idea for expressing join points is using colours for each pointcut and aspect combination, underlying join points with colours.

6.3.4.5 Location of Aspects

Aspects and especially pointcuts can be either located in the aspect package, which models a domain, or in a separate package joining two independent domain packages. The later

approach enables the switching to different pointcut and aspect models and allows developers and modellers to model their domains without knowledge of aspects (Groher and Schulze).

6.3.4.6 Cross Cutting Concern Visualization

Aspect-oriented programming and modelling is about encapsulation crosscutting concerns. A visual modelling framework and visual language possibly needs to give visual feedbacks on which join points are adapted with aspects. Otherwise it is hard for the modeller to debug and correctly model specific pointcuts.

6.3.5 Aspect-Oriented Composition in Model Driven Development

Enabling the use of Aspect-Oriented Modelling (AOM) in a model driven setting includes the definition of formal semantics for aspect composition, as the created (aspect) models have to be processed by automated model transformations. Most AOM approaches define concepts for decomposition, but lack a corresponding composition semantics (Chitchyan, Rashid et al. 2005). The modelling of aspects and their composition can take place at each abstraction layer in an MDA stack, i.e. CIM, PIM, PSM or code (Wampler 2003). Many approaches that deal with aspect model composition propose a composition at the level where aspects are introduced, i.e. mostly at PIM or PSM level. The techniques used for model composition are sometimes called ‘model merging’ and/or ‘model weaving’. In our terminology, *model merging* realises a symmetric composition of models and results in a composed model which constitutes a union of all model elements from the input models.

It is a symmetric composition because of the fact that there is no particular ‘primary’ (or ‘core’) input model, but all input models are equal. Also, all input models as well as the merged model are instances of the same meta-model. Elements from different input models that are matched based on an implicit or explicit matching rule (e.g. by name or meta-attributes) get merged as one element in the output model. Following the terminology of AOP, we see *model weaving* as the asymmetric variant of model composition, because it defines one input model as the primary model, which is adapted by one or more aspect models. The meta-model of the resulting model and the primary model (typically not aspect-aware) are the same, while the aspect model can be based on a different (typically aspect-aware) meta-model. Model weaving also introduces quantification, which allows for 1:n matching of model elements and thus weaving of elements of an aspect model into multiple elements of the primary input model.

Conceptually, model merging and model weaving are specializations of model transformation and can therefore be realised through standard model transformation techniques. In contrast to model transformations as used in an MDA context, model composition generally does not switch abstraction levels or meta-models of the involved models.
A completely different approach beside model transformation is the concept of partial views on one common model repository. This approach is found in most UML-Tools, where each diagram depicts only a part of the model. In this case, no explicit composition step is necessary, because the complete and consistent model is always present in the repository. In this approach, the modularization and separation of concerns would become an issue of the modelling tool that would have to integrate the aspect views dynamically.

We identified 4 variables that describe properties of different model composition approaches:

- ***‘Where’*** - Where are aspects defined and/or composed?
  Possible locations are CIMs, PIMs and PSMs as well as the source code. Most AOM

approaches fit into PIM or PSM level, because they are extensions of the UML. AOM languages representing concepts of a concrete AOP platform should be considered platform-specific, because the underlying aspect composition semantics is dependent on this particular platform.

- ***'When'*** - When is the composition performed?
  Composition can be performed in a horizontal or a vertical transformation step. Horizontal composition means that the composition takes place at either PIM, PSM or Code level. The composed model stays at the same abstraction level and acts as a source for the next transformation steps. A vertical composition takes place at the transition from one abstraction level to the next concrete one, i.e. during a transformation from PIM to PSM or PSM to Code. When model composition can be performed directly at the level where the aspects are modelled or can be delayed to a later point, i.e. a more platform specific level.
- ***'What'*** - What gets composed?
  Symmetric approaches allow the definition of modules that are self-contained and independent of each other. These modules constitute models consisting of structure and behaviour describing one concern. On the other hand, in asymmetric approaches it is often crosscutting behaviour that needs to be integrated in one or more elements of other models. The introduction of additional structure to existing model elements is also possible.
- ***'How'*** - How does the model composition work?
  In the first place, model composition is about matching and integrating structures ("static" model elements) and behaviours ("dynamic" model elements). These are typically identified by name patterns, explicit relationships or meta-data and in the case of behaviours based on control flow, state or events. For asymmetric model composition, the bindings of primary model elements to aspects have to be defined. These bindings can be part of the aspect model or outside of the models. Other possible configuration artefacts for model composition can be constraints and composition directives. The former can further restrict identification and matching of elements from different models, the latter define additional rules for the integration of model elements.

6.4 Quality Assurance in Model-driven Software Development

6.4.1 Introduction

The software industry has a reputation for producing expensive, low-quality software as software systems have reached a level of complexity that puts them beyond our ability to evolve and maintain them easily. This increases the need for software organizations to develop or rework existing systems with high quality.
To improve the quality of their software products, organizations often use quality assurance activities such as refactoring of the source code to tackle defects that reduce internal or external quality aspects of the software. These *quality defects* (i.e., smells, anti-patterns, flaws, bug patterns, pitfalls, etc.) can be diagnosed on the code level but also exist as threats to the quality of earlier abstractions of the software system such as software models.
The techniques to diagnose quality defects (i.e., code smells, anti-patterns, design flaws, etc.) are based upon research from the fields *software refactoring* (Fowler 1999; Simon, Steinbruckner et al. 2001; van Emden and Moonen 2002; Tahvildari, Kontogiannis et al. 2003; Mantyla, Vanhanen et al. 2004; Mens and Tourwe 2004) to diagnose and remove quality defects, *software inspections* (Aurum, Petersson et al. 2002; Ciolkowski, Laitenberger

et al. 2002; Wohlin, Aurum et al. 2002) to manually detect and analyze ambiguities in analysis or coding phases, *source code analysis* (Fenton and Neil 1999; Fenton and Ohlsson 2000) to quantify code characteristics for quality measurement and assurance, and *software testing* (Liggesmeyer 2003) to detect functional defects after implementation.
While some techniques for the diagnosis of quality defects in source code are already known, the diagnosis of quality defects based on architectural information used in model-driven software development (MDSD) and especially platform-independent models (PIMs) from early software development phases are not well understood and open to further investigation. Furthermore, with the rise of MDSD the need for high-quality and maintainable software models will increase. When moving to a completely model based software development approach, the quality of the models from which the applications are generated becomes very important.
In VIDE, quality assurance knowledge for platform-independent models will be researched to increase their quality and ease the development and maintenance of these models. This knowledge will be used to enrich the visualization of the models in order to inform the designers and maintainers about potential threats to model quality.
The remainder of this section describes the academic research base of quality assurance for MDSD with a focus on quality defect diagnosis that is needed in the VIDE research project (especially in WP4). It gives an overview about the core concepts of quality defects and quality defect diagnosis.
A more detailed description of the state of the art and practice for quality assurance in MDSD will be developed in WP4.1.

6.4.2 Quality Defects and Quality Defect Diagnosis

The main concern of software quality assurance (SQA) is the efficient and effective development of large, reliable, and high-quality software systems. While verification and validation efforts in industry typically focus on functional aspects, using techniques such as testing or inspection, other quality aspects are often neglected. However, the non-functional quality of a software product is crucial for its evolution and maintenance by the same or another software developer. Other techniques as software product analysis and measurement are either used to measure software systems and interpret their quality based on a previously defined quality model or to predict project characteristics based on experiences from past measurements. From the deficits found by interpreting the quality characteristics (e.g., software metrics), further actions are derived on an abstract level to improve the software quality.
Another approach in SQA is the diagnosis of explicitly defined defects such as anti-patterns, design flaws, or code smells that represent system-independent defects with a negative effect on a quality such as maintainability. Individual refactorings are used to remove these defects and improve the defective parts without changing its functionality.
The techniques to diagnose quality defects (i.e., smells, antipatterns, flaws, etc.) are mainly based upon research from the field of software refactoring that is very active and beginning to address formalisms, processes, methods, and tools to make refactoring more consistent, planable, scaleable, and flexible (Mens and Tourwe 2004). As Bennett and Rajlich state in their roadmap paper, the central research problem is the inability to change software easily and quickly. Current research issues are being addressed by gathering more empirical information about the nature of software maintenance. The removal of unnecessary complexity is sought through the preservation and management of knowledge for future software maintenance and restructuring of code (Bennett and Rajlich 2000).

6.4.3 Quality Defect Models

Publications including a *description format* of quality defects are given for collections of code smells (Fowler 1999) (Mantyla, Vanhanen et al. 2003), anti-patterns (Brown 1998), design flaws (Riel 1996), design characteristics (Whitmire 1997), or bug patterns (Allen 2002) as well as reengineering patterns (Demeyer, Ducasse et al. 2003). They all define proprietary and different formats for the description of quality defects that are not compatible among each other and neglect information about affected software qualities. There is no comprehensive taxonomy, ontology, or model that helps to classify and distinguish quality defects, their symptoms, and treatments in a uniform way (i.e., similar to the taxonomies in medicine or biology).

Defect classification schemes (Freimut 2001) like ODC are not designed to describe quality defects in a formal, consistent, and complete way. They are designed to support the defect documentation and management and help in the reporting about the software quality, the planning and tailoring of future quality improvement activities (e.g., test planning), and the initiation of preventive measures in early development phases.

6.4.4 Automated Quality Defect Diagnosis Techniques

Currently, several tools were being developed that automatically support parts of the refactoring process. Some of these tools automate the realization of refactorings (e.g., “Extract Method”) – but the detection of places where to apply the refactoring (i.e., quality defects) is still a manual process. Several techniques were developed for code clone detection (Bruntink, van et al. 2004), obsolete parameters or inappropriate interfaces (Tourwe and Mens 2003), and the general processing of source code for of diagnosis and visualization of code smells (van Emden and Moonen 2002).

While some techniques for the diagnosis of quality defects are already known (e.g., the “long method” code smell or several “architectural smells” in the Sotograph tool) techniques for several other quality defects are currently unknown. This is especially true for quality defects that are only diagnosable by analyzing several versions from a software repository.

6.4.5 Software Quality Improvement Techniques

Software Inspections, and especially code inspections, are concerned with the process of manually inspecting software products in order to find potential ambiguities, functional, and non-functional problems (Brykczynski 1999). While the specific evaluation of code fragments is probably more precise than automated techniques, the effort for the inspection is higher, the completeness of an inspection regarding the whole system is smaller, and the number of quality defects looked after is smaller.

Software Testing and debugging is concerned with the diagnosis of defects regarding the functionality and reliability as defined in a specification or unit test case in static and dynamic environments.

Software product metrics are used in software analysis to measure the complexity, cohesion, coupling, or other characteristics of the software product that are further analyzed and interpreted to estimate the effort of development or to evaluate the quality of the software product. Tools for software analysis in existence today are used to monitor dynamic or static aspects of software systems in order to manually identify potential problems in the architecture or sources for negative effects on the quality (e.g., the M-System, ZD-MIS, or the Sotograph). The automated tool-based detection of specific anomalies affecting the quality in software products is relatively rare, to non-existent. Most of these tools (like Checkstyle, FindBugs, Hammurapi, or PMD) analyze the source code of software systems to find violations against project-specific programming guidelines, missing or overcomplicated

expressions, as well as potential language-specific functional defects or bug patterns. Nowadays, the Sotograph can identify architectural smells that are based on metrics regarding size or coupling (Roock and Lippert 2004).

6.4.6 Quality Defect Handling Methods

In addition, the *handling of quality defects and removal activities* in the lifecycle of a software product are not well treated in the literature. For example, the ODC process consists of an opening and closing process for the defect detection that uses information about the target for further removal activities. Typically, removal activities are executed but changes, decisions, and experiences are not documented at all – except for small informal comments when the software system is checked into a software repository.
Software annotation languages used in source code such as JavaDoc or Doxygen can be applied to document the functionality and structure of the software system at the code level. They are tailored for the automated generation of API documents based on a machine-readable syntax. The handling of potential quality defects is not addressed such that accepted quality defects are not presented over and over again and decisions are preserved. Language extensions or mechanisms for machine-readable storing of information about symptoms, defects, or treatments (change history) have not been published.

6.4.7 Beyond the State of the Art

Major parts of this part of VIDE contribute to the fields of refactoring, maintenance, and quality engineering for model-driven software development. The primary contributions to the practice and theory will be:

- A catalogue of existing and the definition of new techniques for quality defect diagnosis. This includes techniques for the extraction, transformation, and integration of information from VIDE-based models to enable model-based quality defect diagnosis techniques.

- A formal model of quality defects on the PIM level that describes quality defects, their structure, symptoms, affected qualities, and associated refactorings as well as their interrelations and dependencies. The model will be usable to classify new quality defects, diagnose quality defects based on identified symptoms, and to configure an optimal treatment (i.e., refactoring) plan.

- An extension of the VIDE platform (based on the eclipse-IDE) for the analysis of software models. It will consists of a plug-in based architecture that is easily extended and adaptable to other modelling languages (with respect to VIDE language extensions), abstraction layers (e.g., other models in MDSD as the CIM), or versioning systems.

6.5 Semantics of Programming Languages from the VIDE Perspective

6.5.1 General Remarks

Semantics determines the meaning of syntactic constructs, that is, the relationship between syntactic constructs and elements of some universe of meanings. This usually understood as referring to the human understanding of meaning and in this case it can be expressed in terms of a natural language. Such semantics we can see in many popular languages, including Basic, C, C++, Java, UML, etc., whose syntactic constructs are explained through more or less

understandable phrases in our everyday expressions or by relationships with other (also informal) constructs. Such semantics, however, are on their own insufficient for a machine, as it is too ambiguous and may contain a lot of unspecified, or poorly specified or inconsistent details. The machine uses algorithmically precise semantics, which in many cases is hard to express using natural language. Such formal semantics are expressed in formal abstract terms, in particular, as mathematical definitions or as actions of a well-defined abstract machine.

The definition of machine-oriented semantics is not as easy because it requires the formal definition of the mentioned universe of meanings and the definition of mappings of the syntax into the universe of meanings. In VIDE we have to take the point of view of application programmers as well as compiler or interpreter designers. The latter point of view requires from us to be precise in specification and sensitive to small machine-oriented semantic details.

The formal actions of the machine and informal understanding of the semantics by application programmers must coincide. A lack of the coincidence is referred to as *semantic reef*. It is a property of the language that most frequently causes application programmers errors due to improper informal understanding of the formal machine behaviour. There are well-known examples of semantic reefs introduced in various languages, for instance, null values or *group by* in SQL (Date 1986; C.J. Date 1992).

The common impression among many professionals is that specification of semantics that is done by people from commercial communities is not sufficiently precise. A logical flaw of such specification is recognized as '*ignotum per ignotum*', i.e. specification of new concepts via undefined concepts (including 'cyclic' definitions). To a great extent, this observation concerns UML, MDA and action semantics/behaviour. To avoid this flaw, there should be clear assumptions which concepts are atomic (undefinable) and how more complex concepts are to be built from the atomic ones. There are approaches in literature that solve this problem of meta circularities (Baar 2003).

The open question concerns if, and to what extent, the VIDE language is to be supported by formal semantics. The discussion below presents various aspects of semantics, pros and cons concerning particular semantic description methods and conclusion concerning some related issues.

6.5.2 What the Description of Semantics is for?

Obviously, users of a language should understand the semantics of language constructs to be aware how to use them in programs and to predict their results. Description of semantics is also necessary for developers and implementers of the language. In general, the description of semantics can pursuit the following goals:

1. Establishing a precise communication channel among the designers and between the designers and implementors of the language.
2. Explaining the meaning and consequences of language's constructs for its users (i.e. programmers, students, etc.). Semantics should be a strong and clear element of the didactic methods concerning the language.
3. Establishing a system of notions that allows one to reason on possible drawbacks, extensions, improvements and new inventions of the language.
4. Formal reasoning on the properties of the language suggests various pragmatic goals: no contradictions, no redundancy, the introduction of new functionalities, query optimization, strong typing, etc.

5. Reasoning on properties and qualities of programs that are manufactured through the language.
6. Standardization: precise, possibly formal specification of semantics is a must. Otherwise standards will be half-standards (as frequently observed), which are more or less compatible concerning concepts and learning the language, but totally incompatible concerning implementation on different platforms.
7. Portability and interoperability: the language may work on different platforms and may access foreign resources. High-level abstract semantics is necessary as a common, platform independent denominator that allows to predict the behaviour of the language constructs on foreign machines or software platforms.

At this stage of the VIDE development, the first two goals of the semantic description are the most important and directly stem from the VIDE's initial description. First, we should find some method of internal communication among VIDE project partners concerning semantics of particular VIDE constructs that we intend to develop. Second, and most importantly, at the end of the project we should prepare a comprehensive manual which presents all aspects of the VIDE use, in particular, its semantics.

6.5.3 Who is the Addressee of the Semantics?

1. Designers of the language, for internal communication during the design.
2. Implementers and testers of the language compiler or interpreter.
3. Future programmers that will use the language; students that will learn the language.
4. Future designers who would like to improve, change or extend the language functionality (compiler, interpreter), to develop some interoperability facilities, including external data, services, libraries, etc.
5. Various research and development staff, including standardization bodies, academic researchers, and so on.

As before, for the VIDE language, the most important are the designers and implementers of the language, as well as future programmers that will use the language. This supports requirement REQ – NonFunc/Semantics 4:

REQ – NonFunc/Semantics 4	Clear and unambiguous notation	**SHOULD**
VIDE should have clear, comprehensible and unambiguous semantic description suited to the users of the VIDE tools		
Description: The VIDE environment should use notation that has clear, comprehensible and unambiguous semantics suited for the user working at the CIM, PIM or PSM level. Therefore, VIDE must offer model views to the user that do not confound the concerns of one level with another (for example, CIM business process description with a PSM sequence model).		

REQ – Semantics 1	Semantics of VIDE Internal Communication	**SHOULD**
Moreover a precise description of the semantics is needed sufficient for internal communication purposes within implementation stakeholders in the development of the VIDE tool.		
This Requirement extends REQ – NonFunc/Semantics 4		

6.5.4 Semantics of Various Features of a Language and of its Environment

Looking at the various functionalities and features of a programming language/environment it is hard to expect that all of them could be described by the same semantic method. Data structures, procedural abstractions, types, classes, query languages, concurrency, exceptions, transactions, etc. may require different approaches to semantic specification. Below we list particular features of programming languages and its environment that could be relevant for VIDE and present some remarks concerning semantic description.

- **Data structures**. Any programming language, including VIDE, must clearly and precisely determine data structures that have to be served by the language constructs. A precise view on data structures is a prerequisite for the description of semantic of any retrieval and manipulation capabilities that act on these structures. This also concerns object-oriented data models. The term 'object' can be understood in myriads of ways, especially concerning how objects, relationships between objects, object hierarchy, object encapsulation have to be represented as machine structures. The specification of VIDE data structures should make clear the attitude to object-oriented models with static (multiple-) inheritance, with dynamic inheritance (dynamic object roles), with encapsulation, with kinds of collections, with semi-structured data (XML), with methods of representing UML associations as data structures, etc. A frequent mistake concerning the semantic model of data structures is that people consider only retrieval, forgetting or neglecting updating, creating, deleting, and other operations on state (which are inevitable for any data structures). On the PIM level, data structures have to be represented in a platform independent way, i.e. without referring to any data storage media. Since VIDE is mainly focussing on UML, data structures are given by a UML class diagram.
- **Persistence of data structures**. The semantic properties and specifications are closely related to the VIDE attitude to persistence and to the assumed application architecture (Atkinson and Buneman 1987). The typical solution is that persistent data is on a shared (database) server, while application logic is on a client. There are a lot of other solutions, in particular, where majority of application (business) logic is on a server. Another dimension for semantic considerations concerns the question if persistent data/objects have the same semantic properties as volatile ones (this is known as orthogonal persistence (Atkinson and Morrison 1995). While such unification makes semantic description much easier, it is not popular in commercial solutions. Persistent data is usually relational and served by SQL, while volatile data is determined by types of some popular (object-oriented) programming language. Such a solution inevitably leads to impedance mismatch concerning many syntactic, semantic and pragmatic features of a language. Concerning the PIM level of the VIDE language, the issue of orthogonal persistence seems to be hard, but the decision must be taken soon, because it may influence the entire project. On the other hand, this issue should not influence too much

the design of the PIM level which is by definition platform independent, and thus independent of concrete persistence techniques.

- **Expressions and queries**. Expressions, such as 2+2, x.sal, A[n+1], sin(x), etc., occur in many programming languages. Expressions usually consist of literals, references to data structures (e.g. variable names), function names and built-in operators. Semantic description of expressions depends on the introduced data structures, allowed literal types and the operators. For instance, complex objects, arrays, collections, pointer links require special syntax of expressions, which (in case of nested expressions) presents some problem with the semantic description. Expressions (in particular) follow the *correspondence principle*, which requires that once some data structures are introduced, there must be complete capabilities of expressions to serve any operation on the structures. In typical programming languages expressions usually do not return bulk output. Such a feature is assigned to query languages. Queries are generalized expressions, but the open question is if VIDE should unify both concepts. Semantics of query languages, especially for object-oriented languages, is not a trivial problem and follows various theories, schools and approaches. Promising approaches to defining semantics of query languages more formally include the stack-based approach (K.Subieta, Kambayashi et al. 1995; Subieta, Beeri et al. 1995; K.Subieta 2004; K.Subieta 2006), with which the PJIIT VIDE team is the most familiar and the most experienced with, or the definition of the OMG's OCL via set-theoretic operations (Richters 2002.; Object Management Group 2003) or meta-modelling techniques (Baar 2003).

- **Imperative program control statements**. Essentially, the MDA action semantics (behaviour) and action language deal precisely with this issue. In the development of VIDE there are the following functional and semantic issues: (1) expressing as much as possible the behaviour through declarative statements (to reduce physical programming overhead, to conceptualize programs, and to make them radically shorter); (2) rising the level of programming granularity through introducing macroscopic statements (acting on collections rather than on individual entities). Both aspects lead to nesting queries within imperative and control statements. Consequences for the language semantics which are triggered by this requirement must be investigated.

- **Procedural abstractions and parameter passing**. As any programming language, VIDE should implement *procedures* and *functions*, with no limitations concerning nested calls and with no limitations concerning recursion. Object-oriented counterparts of procedural abstractions are known as *methods*. The essential difference of methods concerns a bit different name binding policy and encapsulation. Procedural abstractions are usually implemented together with the possibility to declare own local data, which follows the same type system as the entire language. The semantics of procedural abstractions is well-understood and can be easily expressed in terms of operations on a stack-based machine. Procedural abstractions are usually parameterized. There are several well-known parameter passing methods, such as *call-by-value*, *strict-call-by-value* and *call-by-reference*, with operational semantics that is also well-understood. Actual arguments replacing formal parameters are determined by expressions. If all of these or further passing methods are needed remains to be investigated.

- **Types and strong static type checking**. A type system and a strong (compile time and – where possible – edition time) type checking are to be an important feature of VIDE. Types are usually determined by classes/interfaces of a corresponding UML schema, but this could be not enough for typing local (client side) data or objects. Types are intended as constraints on the construction and behaviour of any program entities (in particular,

modules, objects, values, links, procedures, etc.) and constraints on the query/programming context in which these entities can be used. Types are usually considered second-class citizens. Semantics of types is considered a hard problem (Cardelli and Wegner 1985), especially in case of data structures involving object-oriented notions, collections and some irregularities (e.g. cardinality constraints). Roles of the typing system are the following (Stencel 2006): compile-time type checking of expressions/queries, imperative constructs, procedures, functions, methods, views and modules; user-friendly, context dependent reporting on type errors; resolving ambiguities with automatic type coercions, ellipses, dereferences, literals and binding irregular data structures; shifting type checks to run-time, if it is impossible to do them during compile time; restoring a type checking process after a type error, to discover more than one type error in one run; preparing information for query optimization by properly decorating a query syntax tree. (M.Lentner, Stencel et al. 2006; Stencel 2006) report on this issue concerning object-oriented and semi-structured data environments.

- **Classes, interfaces, encapsulation, schemas and meta-models**. In the database community, there are several views on these concepts e.g. (Zdonik and Maier 1990; Cattel 1994; Cattel and Ed 2000; Melton, Simon et al. 2001) while in the object-oriented software-engineering and formal methods communities the view is quite settled (see Section 6.5.6). From the point of view of databases, an attempt to clarify these concepts must be made by assigning pragmatic roles to them: Classes are source code units (second-class citizens) that contain implementation. After compilation they may disappear from the code, or may became special kind of objects employed by the stack-based machine. Interfaces are external specifications of access to objects; they contain no implementation. Types are determined by interfaces, but types can also exist without interfaces. Encapsulation means a special policy concerning scoping, binding and typing, well-known from the object-oriented literature. Schemas are external (application programmer oriented) specifications of a database content and are inevitable pragmatic part of a query/programming languages. A meta-model is, from the database perspective, an internal representation of a schema; it is internally used by the database management system and externally for generic programming with reflection. The above concepts present a definitional knot (especially due to inheritance, dynamic roles, late binding, polymorphism, collections, etc.), having source code incarnation and internal representation. The functionality and semantics of these notions is not obvious, thus should be carefully designed and specified.

- **Exceptions and exception handling, events and event-driven programming**. Exceptions and events are important abstractions allowing to decompose the program control into main and exceptional parts or to decompose the program according to the event-driven programming paradigm. There are several functional (thus semantic) models concerning exceptions and events, for instance, the CORBA model (Object Management Group 2002). Probably, the best strategy for VIDE is to adopt one of them (e.g. the model of Java), together with all the syntactic and semantic consequences.

- **Database abstractions**. The most known database abstractions are virtual database (updateable) views, materialized views and triggers. From the point of view of semantics, the most challenging (but the most useful) are virtual views for object-oriented models (Kozankiewicz, Leszczylowski et al. 2003). An open question concerns if the VIDE project should involve such advanced notions. However, lack of them will much decrease the scope of applications of the VIDE language. The PJIIT VIDE team has proper knowledge and implementation experience concerning virtual updateable object-oriented views.

- **Concurrency, parallelism and transaction processing.** The functionality and semantics of concurrent (parallel) processes/threads implemented in different languages is well understood and presented in many sources. Concerning business processes that are to be built on top of VIDE applications, the feature seems to be important. The problem concerns special syntax for parallel processes, some approach to re-entrant procedures that are required by such processes, synchronization of parallel processes, and issues related to execution of parallel processes on different machines (protocols). The traditional database approach to parallel processes is known as transaction processing in the client-server architecture. This feature also requires special syntax and has many semantic and implementation peculiarities.
- **Aspect-oriented decomposition** (G.Kiczales, Lamping et al. 1997). It is a key feature for creating a truly user–friendly application development environment that could be acceptable for business-oriented programmers. Many difficult features of business applications, such as user logins, security, privacy, transactions, administration, etc. can be implemented as aspects that allow to separate and specialise the application development among many different types of programmers. Weaving aspects with the main code requires some new functionalities (e.g. some ontology on top of the source program structure) and clear semantic description of the weaving processes.
- **Business process abstractions**. This kind of abstractions requires expressing applications in terms of workflows, tasks, resource definition/consumption, business rules (perhaps declarative), monitoring population of executed processes, and so on. Semantics of these abstractions requires to build a meta-model of defined and executed business processes, together with resources such as workflow participants, processed documents and access to external services. Next, it requires some action language, perhaps with nested declarative queries. Defining such functionality on the PIM level may present a challenge, especially if VIDE is to be compatible with some business processes standards.
- **Access to external resources**. Because VIDE is to be a generic and open language (although devoted to particular applications), there should be a clear methodology and functionality of how to connect VIDE to particular external resources (including resources being under the control of a particular operating system). At this stage we can consider several candidates, including CORBA, Web Services and dedicated wrappers (to relational databases, to XML, etc.). This feature requires some predefined programming utilities, which semantics should be clearly specified. VIDE could also follow the service-oriented architecture (SOA) that is postulated in the OMG standardization activity.

The above list of features and semantic description problems can be redundant and/or incomplete, thus will be verified during the further stages of the VIDE project.

6.5.5 Alternatives for Specifying Language Semantics

There are quite a number of approaches used in academia and practice to describe the semantics of languages. Here we list these approaches. In the next section we match them to the needs of VIDE and conclude requirements for the way semantics is defined for the VIDE language.

6.5.5.1 Semantics through Precise Natural Language Descriptions

A very pragmatic approach to language semantics is by defining each language construct in natural language. A popular example for this approach is the Java Language Specification (JLS)(Gosling, Joy et al. 2000), which precisely describes the meaning of the Java language.

Investigations (Stärk, Schmid et al. 2001) have shown that there were initially surprisingly few flaws in that document, considering the size of the document, which have in the meantime been corrected. This example shows that it cannot be excluded that there are flaws in the document nor that the document is complete, since natural language is not formal enough to be understood by formal reasoning systems. However it presents by far the most comprehensible approach and is most suited to users (programmers) of the language.

Everything which can be expressed with some formal model can, more or less precisely, be described by natural language. Important internal structures such as stacks, object stores, meta-models, strong typing rules, event registers, etc., which are necessary to specify precisely the implications of particular language constructs, can all be explained in terms of natural language as the JLS demonstrates. The needed precision is just a matter of discipline.

6.5.5.2 Semantics through Pragmatics

Pragmatics of a language determines its function in interaction between humans or between a human and a machine. Pragmatics describes how to use the language in practical situations, what are the reasons for the use and what goals can be achieved. Pragmatics requires learning how to match expressions of the language to concrete real-life situations, what will be the response from the machine and how we have to interpret the response. Any computer language should be pragmatically efficient, i.e. the language must have the potential to accomplish some important practical goals.

Pragmatics cannot be formalized. It can be expressed in the natural language by general explanations of syntax and semantics, showing some use cases, examples, patterns, anti-patterns, best practices, wrong practices, etc. Majority of user textbooks and documentations of languages are devoted to their pragmatics. However, the only way to teach and learn pragmatics is to use the language for concrete practical situations.

Pragmatics is the most important aspect of any language. Syntax and semantics are important, but only if serve the pragmatic goals of the language. Actually, the description of pragmatics is an inevitable method for specification of semantics for future users of the language. Thus, the basic form of explaining semantics should be a VIDE programmer manual, where all the VIDE functionalities will be explained in the natural language and supported by examples and use cases.

6.5.5.3 Semantics through Implementation

Implementation fully determines semantics. In particular, one can assume that the universe of the meanings is the set of all the sequences of instructions of the Java virtual machine (JVM). The definition of semantics means that all the language expressions are mapped into the set of sequences of instructions of JVM. Such definition of semantics assumes that the meaning of JVM instructions is non-definable or definable by some other—simpler—means.

In this way the definition of the semantics is done by the designers of a compiler or interpreter of the language. This method of semantic definition, however, has disadvantages:

- It must be supported by some informal definition of semantics for application programmers, who rarely understands actions of the Java virtual machine or another assembler-like language.
- Some pragmatic goals of the semantic specification presented in 6.5.2 cannot be achieved. In particular such a semantic description has no meaning in understanding

some basic principles that govern the language, disallows reasoning on the language features (e.g. redundancy, consistency, optimization, extensions, etc.) and promotes ad hoc solutions that could be immature, limited, redundant and inconsistent.

- The semantic specification is dependent on a platform, providing that JVM (however hardware and operating system independent) is a platform too.

Hence, although implementation will ultimately determine the semantics, the VIDE language must be supported by more abstract, platform-independent semantic specification.

6.5.5.4 Semantics through Abstract Implementation

Abstract implementation is a kind of the operational semantics where we have to determine precisely, on the abstract level, all the data structures that participate in query/program processing. Subsequently the semantics of all the language's constructs are defined in terms of some abstract machine acting on these structures. The essential feature of abstract implementation is that it does not refer to any notions of computer platforms, but can be mapped on a 1:1 basis (in a non-optimized version) into a concrete implementation using any programming language, e.g. Java or C++. Abstract implementation shows all the semantic details that are necessary for implementation and allows for deep reasoning concerning various features, including optimizations, non-redundancy, completeness, strong typing, etc. Abstract implementation uses some simple mathematical concepts (sets, functions, relations, tuples, etc.) but does not assume the mathematical method: it appeals to human imagination rather than strives to produce some theorems and formal proofs. An example of the successful application of this method is SQL, where almost all operations (joins, selections, projections, group by) are explained by simple transformations of abstract tables (modelled sometimes as mathematical relations). Concerning database query languages, the abstract implementation method does not need to refer to any mathematical theories such as relational algebras or calculi – all semantic properties can be precisely explained on introduced abstract data structures.

The Stack-Based Approach (SBA) to object-oriented data intensive environments exemplifies this approach (Subieta, Beeri et al. 1995; Subieta, Kambayashi et al. 1995; Subieta 2004; Subieta 2006). Because it deals with full programming environment, abstract implementation of a corresponding query/programming language is more complex and introduces more notions than e.g. relational algebra. In the basic version, to specify run-time operations, it introduces three basic abstract data structures that are well-known in the specification of PLs:

- Object store;
- Environment stack (thus the stack-based approach);
- Query/expression result stack.

These structures are fundamental for precise semantic description of everything that may happen in database query/programming languages. In particular, classical query operators, such as selection, projection, join and quantifiers, can be generally and precisely specified using the above three abstract structures, with no reference to the classical database theories such as the relational/object algebras. Moreover, these structures are also sufficient for explaining the semantics of procedures, functions and methods, including nested and recursive calls, local environment, side effects and various parameter passing methods. The same structures are sufficient to define such database abstractions as virtual views.

Types and strong type checking can also be expressed in terms of abstract implementation. To this end, the following abstract data structures are necessary (Stencel 2006):

- Metamodel (a compile time counterpart of object store); it is a compiled version of a data schema;
- Static environment stack (a compile time counterpart of the environment stack); it stores signatures of run time entities for particular programming environments (e.g. a user session, a database, a currently compiled procedure, a currently analyzed object, etc.);
- Static query/expression result stack (a compile time counterpart of the result stack); it stores type signatures of results that are currently analyzed;
- Decision tables storing type inference rules.

Other functionalities presented in Section 6.5.4 may require other abstract data structures and other abstract machines to process them. For example, processing of exceptions/events requires an abstract event register, aspect oriented decomposition may require structures storing the ontology of the application environment, etc.

The method is very successful for understanding of the semantics by developers of query/programming languages. It can be sensitive to any detail of a data model that we want to consider and to any operation that we would like to introduce. The method is also very efficient for query optimization, strong typing, reasoning on new language's properties, etc. It was successfully applied as a didactic method (to explain the behaviour of particular semantic mechanisms; as well as allowing for fast and painless implementations) and as semantic specification in many projects.

6.5.5.5 Semantics through Mathematical Description

The success of a particular formal semantic specification method is not measured by the fact that somebody has specified formally the language, but by the fact that this specification was efficient: it has achieved pragmatic goals (presented in Section 6.5.2) that cannot be achieved otherwise.

There are many approaches to true formal, that is mathematically based, semantics; these approaches belong to three major classes (Gunter 1992; Reynolds 1998):

- Denotational semantics, whereby each phrase in the language is translated into a *denotation*, i.e. a phrase in some other language; more precisely, one is defining a function from each programming language expression to a value. For real-world languages, this approach turned out not to scale-up.
- Operational semantics, whereby the execution of the language is described directly by describing effects on some notion of state (rather than by translation);
- Axiomatic semantics, whereby one gives meaning to language constructs by describing the *logical axioms* that apply to them; that is, the axioms refine language constructs iteratively into simpler, and eventually atomic ones.

Research on formal semantics of programming languages, has made a lot of progress in recent years and has achieved those goals in Section 6.5.2 which are suited to be tackled with formal language semantics. To give just a few examples from the area on research on the Java programming language (which has received a lot of attention because of its wide-spread use in industry, as well as its relative clean definition):

- There are a number of other approaches for formally defining and extending the Java type system, such as in work by (Müller and Poetzsch-Heffter 2000) and many more

on alias control systems. These approaches cover property 3 of the success criteria outlined in Section 6.5.2.

- (Stärk, Schmid et al. 2001) define the Java semantics formally using abstract state machines (ASMs), a widely acknowledged formal specification method (Börger and Stärk 2003). They prove formally that "*any well-formed and well-typed Java program, when correctly compiled, passes the verifier and is executed on the JVM. It executes without violating any run-time checks, and is correct with respect to the expected behaviour as defined by the Java machine."* The technique thus satisfies property 4 of success as outlined Section 6.5.2.
- (Ahrendt, Baar et al. 2005) specify an axiomatic semantics *Java Dynamic Logic* of Java/JavaCard. With it and an automated theorem prover implemented around it, Java programs can be efficiently formally verified as demonstrated at non-trivial case studies. This technique thus satisfies property 5 of the success criteria as outlined in Section 6.5.2.

These formal approaches have obtained a lot of attention also from big players in software industry. To mention just an example, Microsoft has introduced AsmL/SpecExplorer (Barnett, Leino et al. 2005; Campbell, Grieskamp et al. 2005; Gurevich, Rossman et al. 2005), an executable specification language to describe behaviour which has a semantics founded on abstract state machines.

6.5.6 On the Semantics of the VIDE Language

The VIDE language manipulates instances of UML class diagrams. More precisely, it transfers one instance of a UML class diagram, also referred to here as a snapshot, into another snapshot. Instances of class diagrams - or snapshots - are logically nothing else than a typed first-order structure, where each class is mapped to a type, each association is mapped to a relation, each attribute is mapped to a unary function (Kim and Carrington 1999; Schmitt 2001; Roe, Broda et al. 2003). A side-effect free expression of VIDE is equivalent to an expression in a typed first-order logic. It is interpreted in that structure according to traditional first-order logic.
Structures derived from UML class diagrams are special in the sense that they rely heavily on the abstract data types set, bag, and sequence. It is, however, standard to have these data types included in first-order logic. Basically, interpreting expressions on class diagrams thus reduces to basic set theory. OCL expressions are equivalent to first-order terms and formulas with abstract data types Set, Bag, and Sequence.
An action language adds to such a side-effect free language transitions from one structure to another, or equivalently from one snapshot to the other. First-order structures are thus identified with states. Put differently, the interpretations of the predicates and functions may change though the transition from one state to the other. More precisely, only some distinguished functions and predicates may change interpretation, the so called non-rigid ones, while others, the rigid ones, like mathematical operators, have fixed interpretations in all states. This is the fundamental idea of Kripke structures. This basic idea underpins all established precise modelling notations.
In its most pure form this idea is manifested in abstract state machines (ASMs)(Börger and Stärk 2003), which introduce explicit means to modify structures. ASMs are widely acknowledged as an instrument for precise formalisation of behaviour. If it turns out that VIDE needs formal semantics, ASMs could be a good candidate; because of their good understandability, yet formality, ASMs are increasingly being adopted by industry (see above).

6.5.7 Concluding Requirements on Language Semantics for the VIDE Language

The central goal of VIDE (on the level of modelling languages) is to develop a concrete syntax of an action language suitable to a certain user group. Since defining a formal semantics is not a central part of the project, nor do we aim at any formal reasoning on the language or the language artefacts, we follow a pragmatic approach: We try to define the semantics using natural language (much like in the JLS) as precisely as possible. As soon as it turns out that this is insufficient for any reason, we make use of another well-established method. Which one this is, will be chosen based on the evaluation above as well as on the needs which arise in that situation. This is manifested in the following two requirements:

REQ – Semantics 2	Simple VIDE semantics	**SHOULD**
Keep it simple! As a consequence of Requirement REQ – NonFunc/Semantics 4 (VIDE users are the main target of VIDE language semantics description), after a first analysis it seems sufficient that the semantics of VIDE is described in natural language. If it turns out that language constructs must be defined more formally because we want to apply formal reasoning of any kind in the course of developing the VIDE tool, one should choose a formal method which is well established and state-of-the-art as described above. The method must prove to be as expressible and understandable as abstract state machines.		

7 Standards and Languages

7.1 Introduction

7.1.1 Standards within VIDE

The standards relevant to the project can be assigned into the following categories:

1. **Core modelling standards assumed as a foundation of the VIDE language and platform**. The VIDE language is expected to be based on those standards. As such, the suitability and completeness of the standards with respect to VIDE features will be carefully investigated. Research performed during the project may result in the proposal of extensions or improvements for those specifications. Thus, a high level of external compliance with established standards assumed by MDA is a strong priority.
2. **Related standards that may inspire the solutions of VIDE language and tool**. Other modelling standards, dealing with domains not covered by VIDE, or representing an approach different from the chosen standard base, may inspire solutions of VIDE language.
3. **Standards to be applied in VIDE because of tool interoperability needs or because of the productivity offered by their infrastructure**. The motivation for following particular standards in VIDE software development platform may be twofold. On one hand, software development tools complementary to VIDE may require particular standardized API and / or data exchange formats. On the other hand, tool development may benefit from existing standard-based tools and frameworks.
4. **Platform standards to be supported by model compilers**. The precise set of those standards will be determined by the scope of executable platform mappings chosen to be realised during the project. Apart from this straightforward requirement, analysis of those standardized platforms may allow further conclusions regarding software features that should be abstractly represented on the PIM level.

7.1.2 Technical Requirements

The VIDE project will rely heavily on modelling technology, which is under constant development and set industrially recognised standards. Further, there exist implementations of these standards, which can be used by the project. Modelling tools are normally being built upon a modelling infrastructure. This infrastructure provides the substrate, containing basic services like persistency, transactions and commands. The services provided by the modelling infrastructure are not necessarily limited to basic services, but can also contain services like model transformations and frameworks for graphical modelling.

This chapter provides a survey of requirements that the VIDE project posses on the modelling infrastructure. These requirements are not necessarily bound to, or solved by, a single standard. Nevertheless, it is not VIDE project goal to build a modelling infrastructure from the ground up. Therefore, existing standards should be used where possible, providing these do not endanger the overall project goal. Utilization of common standards provides manifold benefits for the project. First, the adoption and dissemination of project's results is easier, if the compatibility to common standards is preserved. Second, the project itself can benefit from usage of common industrial tools by adhering to common standards. By following standards (at least for the first argument) it does not necessarily mean adherence to formal

compliance. The compatibility can be preserved by usage of standard's concept in the large, while deviating from it in the details.
Technical requirements are addressed in this chapter in Section 7.3. There we provide an overview of features required by the VIDE project, in regard to basic modelling infrastructure. Furthermore, this section provides an overview of the existing modelling standards. Section 7.4 provides the requirements of the project on model transformation standards and describes the state of the art model transformation techniques. Section 7.5, describes the need for a graphical modelling framework and concludes with the overview of Eclipse foundation's GMF framework.

7.1.3 Requirements of Modelling Technique on CIM-Level

The models on CIM level can be used as requirements definition. They serve as interface to the users of software which should be developed. Therefore the domain user has to understand the CIM models. This means he must at least be able to validate them. According to this, the modelling technique to be developed within the VIDE project has to take the business view of domain users into account.
As VIDE follows the MDA approach, at least some of the CIM level models must be transformable into PIM level. Additionally, some PIM level models must be transferable back into CIM level models. This is necessary as changes on code level should be propagated back to the levels above, which is required for the vision of the project to enable software development on code and model level.
For testing and validation purposes, it should be documented how the developed software will be used in a business process, because this helps to reveal requirements for the software development and makes it easier for the domain user and the developers to validate the software.
As one of the major ideas of VIDE is rapid application development (RAD), that includes many feedback cycles with potential users. This potentially reduces costs of development failures because later changes to the software design are more expensive. This has to be taken into account in the development of the VIDE procedure model for software development.

7.1.4 Enterprise Frameworks and Architectures

This section gives an overview of three commonly used modelling frameworks. Modelling frameworks usually consists of several perspectives and can be used in a software development process on different levels, from the end users view right up to the implementation level. This division of the modelled artefacts in different perspectives and the different languages (which are used for each perspective) gives an idea of what subjects have to be considered in the development of the VIDE language, especially on CIM level.

We define a framework as a fundamental structure which allows the definition of the main sets of concepts that support the modelling and development of an enterprise. The dominant enterprise modelling frameworks and architectures that are being pursued by industry and interest organisations are:

1. The Zachman Framework
2. ARIS (Architecture of Integrated Information Systems)
3. The CIMOSA Framework

The Zachman Framework has been used by the Canadian Government to design their enterprise architecture approach, and as a reference categorisation structure for enterprise knowledge repositories.
ARIS (Architecture of Integrated Information Systems) has strong, top-down process modelling and integration capabilities, but lacks expressiveness for other aspects and the big picture created by a holistic approach.
The CIMOSA Framework is a good reference framework, but lacks expressiveness for multiple dependencies between types of view, for evolving concepts, contents and capabilities and for capturing context.
Common to all of these is that they are descriptive frameworks, defining enterprise domains and their views and contents.

7.1.4.1 The Zachman Framework for Enterprise Architecture

The Framework (Zachman 1987; Sowa and Zachman 1992) as it applies to enterprises is simply a logical structure for classifying and organising the descriptive representations of an enterprise that are significant to the management of the enterprise as well as to the development of the enterprise's systems. Moreover it offers the basic structure for the organisation, access, integration, interpretation, development and change of the representations of objects within an enterprise.

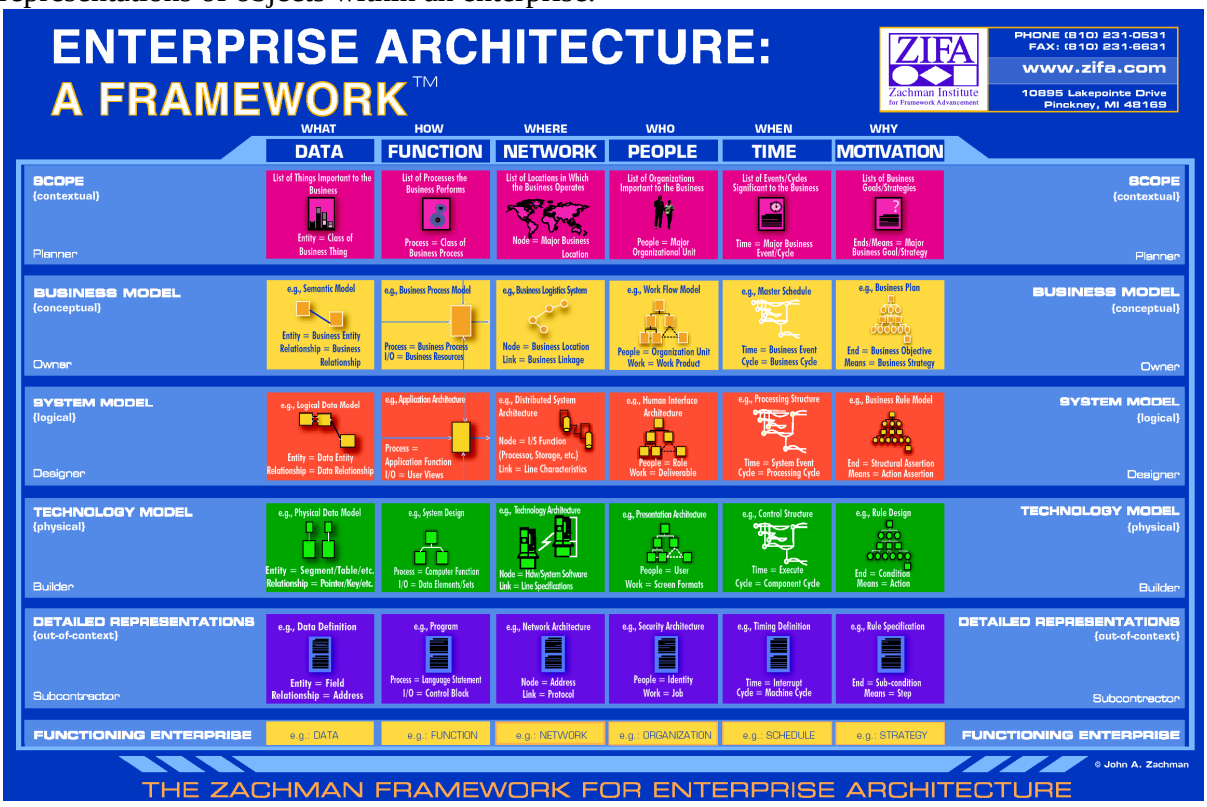


Figure 4: The Zachman Framework (http://www.zifa.com/framework.pdf)

The figure above shows that the framework is designed as a matrix. On the horizontal axis six different views are presented. To each of this views a "W"-question is assigned. The views are:

1. Data (What)
2. Function (How)
3. Network (Where)
4. People (Who)

5. Time (When)
6. Motivation (Why)

The vertical axis is defined by five different layers and the organizational roles of the people working on that layer. The layers are

1. Scope
2. Business Model
3. System Model
4. Technology Model
5. Detailed Representations

The roles that are assigned to these five layers from Scope to Detailed Representations are Planer, Owner, Designer, Builder and Subcontractor. So by both axes 30 different model types are distinguished.
In general the Zachman Framework is domain independent. It has less the character of a procedure model than more the character of an instrument for project management, that ensures all important aspects of an enterprise are covered. The framework doesn't provide specific languages for the representations. Therefore it's not very much formalized, which decreases the ability of IT support.

7.1.4.2 ARIS (Architecture of Integrated Information Systems)

ARIS has been developed by Scheer at the University of Saarbruecken (Scheer 1999). The conceptual design of the Architecture of integrated Information Systems (ARIS) is based on an integration concept which is derived from a holistic analysis of business processes. The first step in creating the architecture calls for the development of a model for business processes which contains all basic features for describing business processes. The result is a highly complex model which is divided into individual views in order to reduce its complexity. The enterprise modelling approach of ARIS (Scheer 1999) divides enterprise models into five individual views as shown in Fig. 5:

- **Data View**
- **Function View**
- **Organisational View**
- **Output View**
- **Control View**

While the data, functional, organisational and output view represent an isolated aspect the control view integrates these four other views. The control view represents the steps of a business process and the control flow between them, which includes sequences, forks and joins.
The integration of the other views is done by annotating objects of the other views to parts of the control flow. For example an entity type from the data view and an organisational unit from the organisational view can be related to a function in the control view, to express that this function is carried out by the organisational unit assigned and accessing the data object represented by the entity type. The most common language which is used in the control view of ARIS is the Event Driven Process Chain (EPC), which is regarded in section 7.2.3.1.

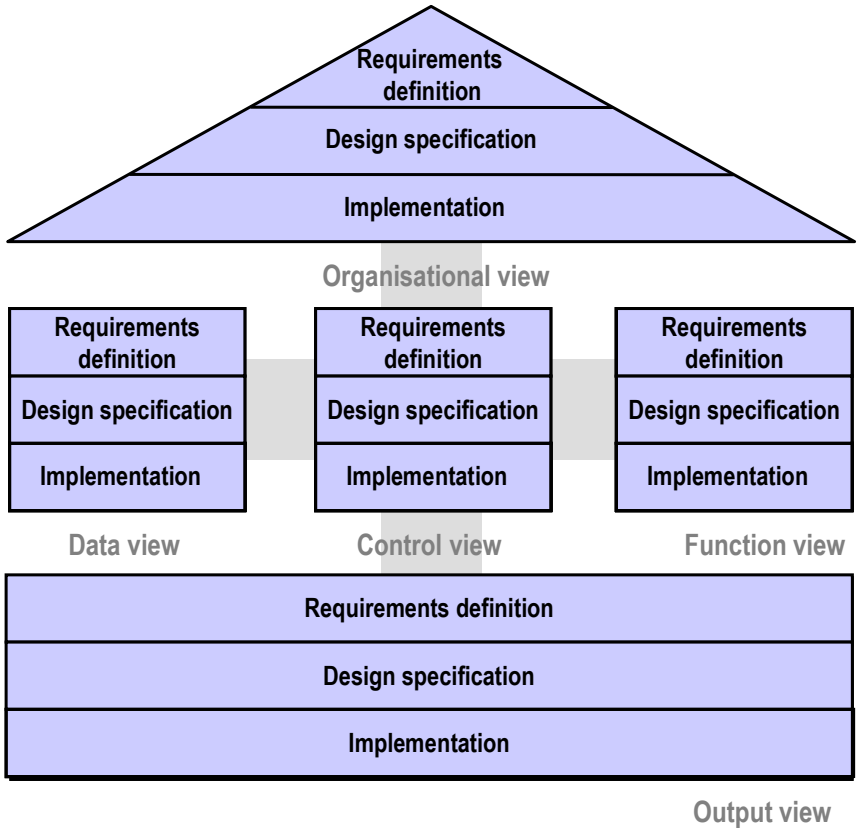


Figure 5: View concept of ARIS

Due to this division, the contents of the individual views can be described by special methods which are suitable for this view without having to pay attention to the numerous relationships and interrelationships with the other views. Afterwards, the relationships between the views are incorporated and combined to form an overall analysis of process chains without any redundancies. A second approach that also reduces the complexity is the analysis of different descriptive levels:

- **Requirements definition**
- **Design specification**
- **Implementation**

Following the concept of a lifecycle model the various description methods for information systems are differentiated according to their proximity to information technology. This ensures a consistent description from business management-related problems all the way down to their technical implementation. Thus, the ARIS architecture forms the framework for the development and optimisation of integrated information systems as well as a description of their implementation. In this context, stressing the subject-related descriptive levels results in the ARIS concept being used as a model for creating, analysing, and evaluating business management related process chains.
The ARIS concept was designed at the application independent meta level (the different levels are: instance level, type level, meta level and meta² level) and is based on a process-oriented approach. Due to the fact that terms permissible at this level are also valid for underlying application types and instances, the ARIS concept is automatically applied to underlying modelling levels as well.

7.1.4.3 CIMOSA

CIMOSA (Computer Integrated Manufacturing open System Architecture) was developed as an architecture for describing CIM systems by several EU projects within the ESPRIT framework. It aims at offering an architecture and methods for creating common CIM modules which can be plugged together to form a new, customer-oriented application system. The framework is based on the so-called CIMOSA cube, showed in the figure below.

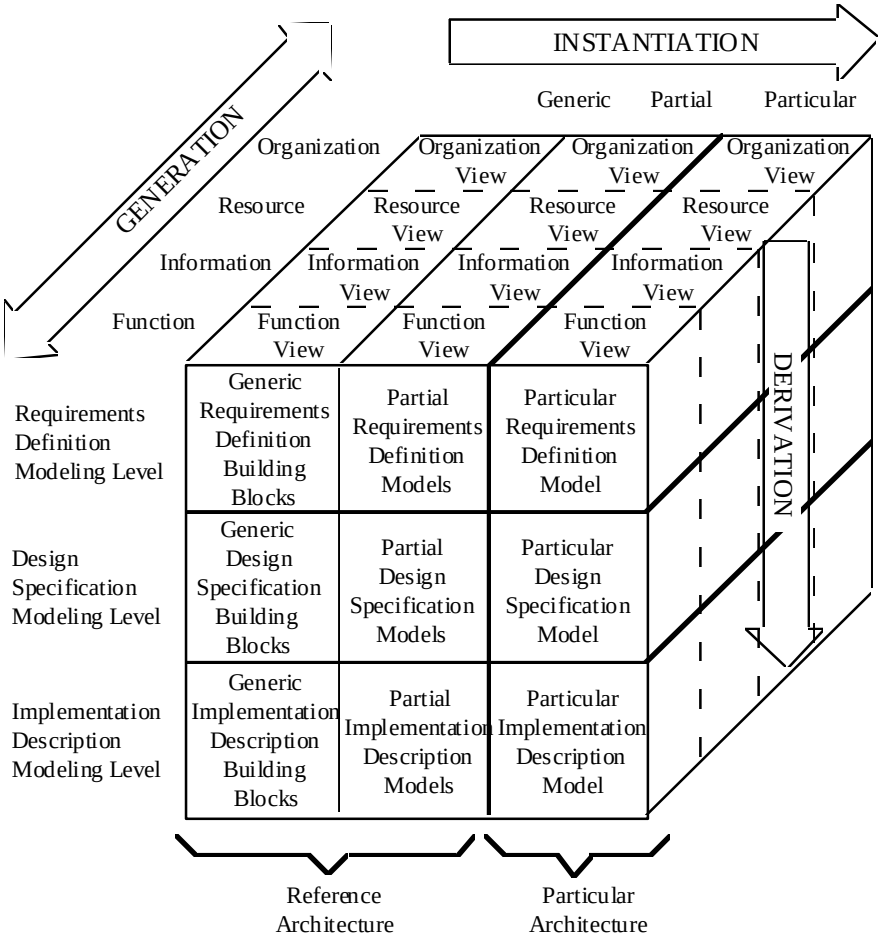


Figure 6: The CIMOSA cube (Vernadat 1996)

The three axes of the cube describe the three dimensions of CIMOSA, the “Stepwise Derivation”, the “Stepwise Instantiation” and “Stepwise Generation”. The vertical direction derivation defines three conceptual layers beginning with the requirements definition level over the design specification modelling level to the point of implementation description modelling level. The horizontal arrow instantiation shows the refinement of fundamental requirements of a system to industry branch specific ones and ends with company specific concerns. The third dimension “Stepwise generation” fragments the perception of an information system into different views. There are four views that compose this dimension, the “Function View”, the “Information View”, the “Resource View” and the “Organisation View”. They are used to describe different modelling aspects like activities, events, processes, data definitions, production related resources and organizational structures.
The CIMOSA architecture was designed the offer systematic ways to include reference models into the description of information systems up to software generation processes.

7.2 Modelling Standards

7.2.1 UML 2

REQ – Lang 1	Usage of UML2 Behaviour ("Action Semantics")	**SHOULD**
VIDE should use the behavioural model elements of UML2 (earlier known as "UML Action Semantics"), unless proven insufficient.		
Description: As decided by partners and as consequence of REQ – Lang 4.		

7.2.1.1 Introduction

The current version of UML (UML 2.0) brought a significant reorganisation of the specification. The document has been divided into two complementary specifications: UML 2.0 Infrastructure (Object Management Group 2004) and UML 2.0 Superstructure (Object Management Group 2004). The former groups the fundamental notions of the language (most of them considered abstract) that are intended to form a universal common base not only for different parts of UML or their specialisations, but also for meta-models of different object-oriented modelling languages. This allowed the removal of redundancy and alignment among the modelling languages already specified by OMG by that time: UML, CWM (Common Warehouse Meta-model) and MOF (Meta Object Facility). When the development of a completely new language is considered, the advantage of the application of the core constructs is twofold. First, they may be suitable to adopt unchanged as a core of the meta-model of such a language, and second, a similar set of constructs are available as a means of the meta-model definition. Further discussion of this issue is placed in the section devoted to MOF.

Such architecture promotes extensibility, reusability, customizability and evolvability and makes it easier do develop standard compliant tools. At the same time however, the compliance criteria may become fine-grained, as they may refer to particular parts and levels of the specification. For example, one of those criteria distinguishes the following levels of compliance:

- abstract syntax compliance,
- concrete syntax compliance,
- abstract syntax with concrete syntax compliance,
- abstract syntax with concrete syntax and diagram interchange compliance.

Another characteristic of the new version is a high modularity of its specification. To achieve it, the pre-existing notion of package was extended with a new kind of relationship – namely package merge. Previously, reuse of common core meta-model constructs was possible through stereotyping or by subclassing. When using package merge, the merging package may define classes of the same name as the classes existing in the package being merged. The result is a merging package with classes, whose specification combines base and newly created features in an analogous way as in case of subclassing. Different merges of a given package do not affect its contents, thus there is no interference among packages created as separate merges. This style of meta-model definitions is accompanied with extensive use of derived features – especially through subsetting and unions among association ends across the inheritance hierarchy. This offers a significant flexibility and simplifies particular views of the language (shallower meta-class hierarchies). However, it also makes it more difficult to grasp a whole meta-model. Now, apart from knowing a name of a given UML meta-class we are interested in, it is necessary to choose among several versions of its definition created by package merges.

From the MDA point of view, it is important to note the improvements in the means of profile definition. This should be helpful especially for easier specification of platform-specific model (PSM) profiles.

7.2.1.2 UML for Behavioural Specifications

One of the key areas of the analysis of the UML standard is behaviour specification. Since UML 1.4, the high level behavioural constructs (like state machines, interactions, use cases) are accompanied with 'action semantics' allowing specific and precise computational steps. *Actions*, which significantly contribute to the overall meta-model size deal with atomic tasks including the following manipulations:

- behaviour invocations,
- parameter passing,
- object creation / deletion,
- link manipulation creation / deletion (including support for links instantiating n-ary or qualified associations or association classes),
- reading extent and object reclassification,
- variable manipulation.

Writing objects into durable storage is also explicitly represented. The actions are elementary in terms of UML elements processed – thus their complexity may vary (e.g. an assignment of primitive value vs. link object creation).
To pass data between actions, a notion of (input or output) *pin* was introduced. A pin is a typed element that may be assigned a value.
Actions can be included within activities, which provide e.g. control structures (conditionals, loops) and data flows. Activities in turn are generalized into *Behavior* that can be attached to various higher level UML behavioural constructs. In particular, it can serve as an implementation of a *Behavioral Feature* owned by a *Classifier* (which covers e.g. class-defined object operations).

In the following, we give a brief overview of the currently standardised UML notions capable of handling executable modelling of business applications, with the focus on those of them that cover the core notions known from popular programming languages. Its purpose is to provide a starting point for the definition of the standardised base of the VIDE project, which is intended to:

- guarantee a proper level of abstraction for expressing data manipulations and mapping them onto the implementation platform, and
- provide a common representation, which would reuse standardized solutions at the model compiler side and allow for developing various (yet compatible) syntactical solutions for the platform independent model.

7.2.1.2.1 Actions as a Part of UML Behaviour Model

The UML 2 notions serving for behaviour modelling are organized basically into the following four units, each defining the elements founding one of the main behaviour modelling UML diagrams:

- Actions
- Interactions
- StateMachines
- UseCases

The constructs reusable among those models are grouped into the package CommonBehavior.
To support modelling of the details of behaviour, another package Actions was added, whose contents is directly related to the notions of the Activities package.

Action nodes, control nodes and object nodes are used for activity specification.

The specification (Object Management Group 2004) calls Action the fundamental unit of behaviour and states that: *"Actions are contained in behaviours, which provide their context."* Actions include operation calls, signal sends, direct behaviour invocations. It also states:
"A primitive action either carries out a computation or accesses object memory, but never both.
A surface action language would encompass both primitive actions and the control mechanisms provided by behaviours. In addition, a surface language may map higher-level constructs to the actions.
However, in the execution of actions the lower multiplicity bound is ignored and no error or undefined semantics is implied. (Otherwise it is impossible to use actions to pass through the intermediate configurations necessary to construct object configurations that satisfy multiplicity constraints.)"
Due to their central role, we present a complete list of UML 2 actions, grouping them into several groups based on their purpose.

7.2.1.2.2 Action Hierarchy Overview

The classes described in the Actions chapter of the UML Superstructure are enumerated here to show their generalization hierarchy and group them according to their purpose (the indents and arrows depict the generalization-specialization relationship among Action classes).

Invocation actions:
InvocationAction
- ← (CallAction
  - ← CallBehaviorAction,
  - ← CallOperationAction),
- ← SendSignalAction,
- ← BroadcastSignalAction,
- ← SendObjectAction)

StartClassifierBehaviorAction

Object actions:
CreateObjectAction, DestroyObjectAction, TestIdentityAction, ReadSelfAction

Structural feature actions:
StructuralFeatureAction
- ← ReadStructuralFeatureAction,
- ← (WriteStructuralFeatureAction
  - ← AddStructuralFeatureAction,
  - ← RemoveStructuralFeatureAction,
  - ← ClearStructuralFeatureAction)

Link actions:
LinkAction
- ← ReadLinkAction,
- ← (WriteLinkAction
  - ← (CreateLink
    - ←CreateLinkObject),
  - ← DestroyLink),

ClearAssociation

Value processing actions:
ValueSpecificationAction
ReduceAction

EventActions:
AcceptEventAction
← AcceptCallAction
ReplyAction
UnmarshallAction
RaiseException

Classifier-related actions:
ReadExtentAction
ReclassifyObjectAction
ReadIsClassifiedObjectAction

Variable actions:
VariableAction
← ReadVariable,
← (WriteVariable
← AddVariableValue,
← RemoveVariableValue),
← ClearVariable

7.2.1.2.3 Common Elements of the Actions Unit

Below short descriptions of the Action meta-classes based on the UML Superstructure specification are provided. Where it is needed to clarify a given action's purpose, the description mentions the attributes or associations specified for that action meta-class.

7.2.1.2.3.1 Action

The execution of an action represents some transformation or processing in the modelled system, be it a computer system or otherwise.

7.2.1.2.3.2 Pin

A pin is a typed element and multiplicity element that provides values to actions and accept result values from them.

7.2.1.2.3.3 InputPin

An action cannot start execution if an input pin has fewer values than the lower multiplicity. The upper multiplicity determines how many values are consumed by a single execution of the action.

7.2.1.2.3.4 ActionInputPin

An action input pin is a kind of pin that executes an action to determine the values to input to another. It indicates the action used to provide values. Cf. InputPin.
ActionInputPin is introduced to pass values between actions in expressions without using flows.

7.2.1.2.3.5 OutputPin

Holds output values produced by an action.

7.2.1.2.3.6 ValuePin

It provides a value by evaluating a value specification. ValuePin specifies the value using ValueSpecification metaobject.
It is introduced to provide the most basic way of providing inputs to actions.

7.2.1.2.3.7 ValueSpecificationAction

It evaluates value specification.

7.2.1.2.3.8 OpaqueAction

This class serves for representing an action with implementation-specific semantics (the classes of similar nature are also OpaqueExpression and OpaqueBehavior).
The body of opaque action and the language used in it are specified as two string attributes.

7.2.1.2.3.9 ReduceAction

The behaviour is invoked repeatedly on pairs of elements in the input collection. Each time it is invoked, it produces one output that is put back in an intermediate version of the collection. This repeats until the collection is reduced to a single value, which is the output of the action. The ordering of that processing can be enforced with the attribute isOrdered.

7.2.1.2.4 Invocation actions

7.2.1.2.4.1 InvocationAction

(abstract) InvocationAction covers various actions invoking behaviour. It specifies arguments to be used in the invocation.

7.2.1.2.4.2 BroadcastSignalAction

It broadcasts asynchronous message (using a Signal object).

7.2.1.2.4.3 CallAction

(abstract) CallAction covers actions that invoke behaviour and receive return values. It can be declared as synchronous – by default, or asynchronous. Also specifies the call result.

7.2.1.2.4.4 CallBehavior

It specializes CallAction. Represents a call action that invokes behaviour directly rather than invoking a behavioural feature (e.g. Operation) that, in turn, results in the invocation of that behaviour.

7.2.1.2.4.5 CallOperationAction

It specializes CallAction. Represents an action that transmits an operation call request to the target object, where it may cause the invocation of associated behaviour.

7.2.1.2.4.6 RaiseExceptionAction

It causes an exception to occur. The input value becomes the exception object.

7.2.1.2.4.7 SendObjectAction

With this action, a request object is being sent asynchronously to the specified recipient.

7.2.1.2.4.8 SendSignalAction

Signal is being sent asynchronously to the specified recipient.
Creates a signal instance from its inputs, and transmits it to the target object, where it may cause the firing of a state machine transition or the execution of an activity.
It requires specification of the type of the signal to construct and of the target object.

7.2.1.2.4.9 StartClassifierBehavior

Specifies an explicit initiation of the classifier behaviour. The behaviour object is provided as input.

7.2.1.2.5 Object actions

7.2.1.2.5.1 CreateObjectAction

Performs just a creation of a new object of a given classifier.
In particular, no behaviours are executed, no initial expressions are evaluated, and no state machine transitions are triggered. The new object has no structural feature values and participates in no links.

7.2.1.2.5.2 DestroyObjectAction

It denotes object destruction. Optionally can indicate the destruction of object links and owned objects (by default – false).

7.2.1.2.5.3 TestIdentityAction

Compares two objects and returns the result of their identity comparison.

7.2.1.2.6 Structural feature actions

7.2.1.2.6.1 StructuralFeatureAction

(abstract) StructuralFeatureAction provides details common for various manipulations of object structural features. It specifies the object and its feature to be manipulated.

7.2.1.2.6.2 ReadStructuralFeatureAction

It retrieves the value of a structural feature.

7.2.1.2.6.3 WriteStructuralFeatureAction

(abstract) WriteStructuralFeatureAction modifies a structural feature by applying the value provided. It specifies the value to be added or removed.

7.2.1.2.6.4 AddStructuralFeatureAction

Adds (and potentially replaces) feature value of a given object.

7.2.1.2.6.5 ClearStructuralFeatureAction

Removes all values of a structural feature.

7.2.1.2.6.6 RemoveStructuralFeatureValueAction

It removes a value of a structural feature. To indicate the value to be removed specifies removeAt and isRemoveDuplicates attributes.

7.2.1.2.7 Link actions

7.2.1.2.7.1 LinkAction

(abstract) It represents a link manipulation.

7.2.1.2.7.2 ReadLinkAction

It returns linked objects at the end not specified by the inputs (for this purpose the information as specified in the LinkAction is provided).

7.2.1.2.7.3 ReadLinkObjectEndAction

Given a link object and link end, retrieves the object at that end.

7.2.1.2.7.4 ReadLinkObjectEndQualifierAction

Retrieves qualifier value at the given end of the given link object.

7.2.1.2.7.5 WriteLinkAction

(abstract) Creates or destroys links.

7.2.1.2.7.6 CreateLinkAction

CreateLinkAction creates a link or a link objects. It provides no return data. Supports also ordered associations. The objects of LinkEndCreationData are used as the input.

7.2.1.2.7.7 CreateLinkObjectAction

Specialization used for instances of association classes.

7.2.1.2.7.8 DestroyLinkAction

Destroys links or link objecs. The objects of LinkEndDestructionData are used as the input.

7.2.1.2.7.9 ClearAssociationAction

Destroys all links of an association in which a particular object participates.
ClearAssociationAction is introduced to remove all links from an association in which an object participates in a single action, with no intermediate states where only some of the existing links are present.

7.2.1.2.7.10 LinkEndData

It identifies one end of a link to be read or written by the children of LinkAction.
The purpose of this element is described in the specification in the following way:
"A link cannot be passed as a runtime value to or from an action. Instead, a link is identified by its end objects and qualifier values, if any. This requires more than one piece of data, namely, the statically-specified end in the user model, the object on the end, and the qualifier values for that end, if any. These pieces are brought together around LinkEndData. Each association end is identified separately with an instance of the LinkEndData class."
As can be seen, this approach to link identification is a consequence of the decisions made for the UML data model: the lack of a link's own identity and the support for n-ary associations and links.
LinkEndData specifies the end Property and the object for the given end.

7.2.1.2.7.11 LinkEndCreationData

As inherited, plus isReplaceAll, insertAt provide the link placement details.

7.2.1.2.7.12 LinkEndDestructionData

As inherited, plus isDestroyDuplicates, destroyAt provide the link destruction details.

7.2.1.2.7.13 QualifierValue

QualifiedValue is used to specify actions on links of qualified associations.

7.2.1.2.8 Accept event actions

7.2.1.2.8.1 AcceptEventAction

It waits for the occurrence of an event meeting specified condition.
It serves for handling asynchronous messages (time event, change event, signal event).

7.2.1.2.8.2 AcceptCallAction

It Denotes the receipt of a synchronous call request.
In addition to the normal operation parameters, the action produces an output that is needed later to supply the information to the ReplyAction (necessary to return control to the caller). This action is for synchronous calls. If it is used to handle an asynchronous call, execution of the subsequent reply action will complete immediately with no effects.
The return information value is opaque and may only be used by ReplyAction.

7.2.1.2.8.3 ReplyAction

ReplyAction specifies: returnInformation, replyValue, replyToCall (the latter is the instance of Trigger class).
The execution of a reply action completes the execution of a call that was initiated by a previous AcceptCallAction. The two are connected by the returnInformation value, which is produced by the AcceptCallAction and consumed by the ReplyAction.

7.2.1.2.8.4 UnmarshallAction

It breaks an object of a known type into outputs each of which is equal to a value from a structural feature of the object. It has been introduced to read all the structural features of an object at once.

7.2.1.2.9 Classifier Actions

7.2.1.2.9.1 ReadExtentAction

It retrieves the current existing instances of a classifier.
In the description of this element again we find an important remark on the store model of UML objects.
"*It is not generally practical to require that reading the extent produce all the instances of the classifier that exist in the entire universe. Rather, an execution engine typically manages only a limited subset of the total set of instances of any classifier and may manage multiple distributed extents for any one classifier. It is not formally specified which managed extent is actually read by a ReadExtentAction."*

7.2.1.2.9.2 ReadIsClassifiedObjectAction

It determines whether a runtime object is classified by a given classifier.
(Also specifies whether the classification is direct.)

7.2.1.2.9.3 ReadSelfAction

It retrieves the host object of an action.

7.2.1.2.9.4 ReclassifyObjectAction

For the object indicated in the instance of this action, one or more of its classifiers are changed.

7.2.1.2.10 Variable actions

7.2.1.2.10.1 VariableAction

(abstract) VariableAction generalizes actions dealing with statically specified variables. It specifies the variable to be accessed.

7.2.1.2.10.2 ReadVariableAction

It retrieves variable value.

7.2.1.2.10.3 WriteVariableAction

It modifies variable by applying the value provided.

7.2.1.2.10.4 RemoveVariableValueAction

It removes one variable value. Specifies the way the removal should be performed: removeAt and isRemoveDuplicates determine the detailed effect of this action.

7.2.1.2.10.5 ClearVariableAction

ClearVariableAction is a variable action that removes all values of a variable.

7.2.1.2.10.6 AddVariableValueAction

It supports multi-valued and optionally ordered variables.
"AddVariableValueAction is introduced to add variable values. isReplaceAll is introduced to replace and add in a single action, with no intermediate states of the variable where only some of the existing values are present."

7.2.1.3 Common behaviours

The package defines common and generalized elements used in various behaviour models. Several characteristic elements of that unit that may be important for VIDE are enumerated here.

7.2.1.3.1 OpaqueBehavior

Its semantics is determined by implementation. OpaqueBehavior may be specified in one or more languages as indicated by its attributes:
• body – String attribute specifying the behaviour in one or more languages.
• language – String attribute specifying the languages the body (in the same order as the body strings).

7.2.1.3.2 FunctionBehavior

(specializes OpaqueBehavior) It represents an opaque behaviour that does not access or modify any objects or other external data. The specification says: *"Specific primitive functions are not defined in the UML, but would be defined in domain-specific extensions. Typical primitive functions would include arithmetic, Boolean, and string functions."*

7.2.1.3.3 OpaqueExpression

Expression defined by Behavior, which is required to return exactly one result (single value or single set of values).

7.2.1.3.4 Trigger

A trigger specifies an event that may cause the execution of an associated behaviour.
It may also specify one or more ports (Port is newly added meta-class of UML 2) for an event which implies that the event triggers the execution of an associated behaviour only if the event was received via one of the specified ports.

7.2.1.4 UML Activities vs. Data-oriented Applications

Together with the Actions unit, the Activities form a foundation for executable model behaviour specification.
This package needs thus to be investigated in order to assess if the current shape of UML Behavior (with its direct support focused around the core notions of a typical programming language) allows pure UML models using it to completely and adequately specify database-based applications on the platform independent level.
The activity modelling elements were designed (and partitioned into appropriate sub-packages) to support two main styles of behaviour specification:

- flow-based (useful e.g. for process modelling), where the sequence of activities and actions is restricted by data flow dependencies (the mechanism of execution can be characterized as push style)
- structured (useful for software modelling), typical for traditional programming languages, including loops, control flows, variable access and traditional conditional instructions (allowing for rather pull-style specification).

While the latter seems to be the most straightforward to be used by VIDE, the flow-based style also remains useful, e.g. due to its importance for business process modelling area and because of its looser approach to action sequencing.
Moreover, UML allows for combining those solutions in a single model.

7.2.1.5 Core Notions of the Activity model

Apart from the main behaviour units: Actions and Activities, the following notions were introduced to describe the behaviour logic and data handling.

ObjectNode contains value at runtime. It realises object flow in an activity.
Object Node represents an instance of particular classifier available at a particular point in activity.
The following kinds of object nodes occur:

Pin, Activity Parameter, Central Buffer, Data Store

Although it is not defined as a specialization of *MultiplicityElement,* it assumes storing multiple instances and specifies their allowed number in another way (with an *upperBound* attribute). It also allows to specify a selection and ordering logic for instances outgoing to be consumed by subsequent activities. The selection behaviour cannot have side effects.
The instances/values stored in an *ObjectNode* are called tokens.
Dealing with many tokens of *ObjectNode* in the same activity is distinguished from the situation where the tokens appear in the context of a different execution of a given activity.
The selection behaviour can be overridden by the selection behaviour specified for an outgoing edge (by *ObjectFlow*).

Pin – input / output node for an action (provides values and accepts results).
It specifies type and multiplicity of values to be processed by actions.

ActivityParameterNode – input /output node for activities. It contains a *Parameter* instance.

Parameters (as defined in the Kernel package) are extended in Complete Activities with streaming, exceptions and parameter sets.

Variable serves for passing data between actions indirectly.
It is local to a structured activity group.

CentralBufferNode is described as an object not tied (in contrast to other object nodes) to action (like *Pin* is) or activity (like *ActivityParameterNode* is) but passing data flows between them and supports flows from multiple sources and destinations. It serves as a buffer for multiple in-flows and out-flows.
*CentralBufferNode*s give additional support for queuing and competition between flowing objects.

DataStoreNode is a specialization of *CentralBufferNode* which keeps incoming tokens persistently (that is, tokens moving downstream are copied rather than removed).
Data kept in *DataStoreNode* is persistent and used when needed in contrast to data from a regular *CentralBufferNode* which is volatile and used when available.

This seems to be the only place where UML specification explicitly refers to persistent storage. However, the specification of the DataStoreNode notion reveals that only some of its properties are common with the analogous notion from the database technology area and that this is thus not suitable as the base of database management modelling.
DataStoreNode specializes CentralBufferNode, which in turn specializes *ObjectNode*. This means that DataStoreNode is just a constituent of object flow and as such owned by particular instance of activity execution (and therefore lasts only within its lifetime).
Thus, as commented in (Bock 2004): *"UML data store nodes are still active and transient, however, and do not completely capture pull semantics."*
Moreover, a number of further platform neutral properties seem necessary to describe an application dealing with a persistent data source. The related issues that need to be investigated in the context of such a model include: data identification, access privileges, transactional constraints, expressing higher level data manipulation statements and other.

The relations among the key meta-classes related with the Activities elements discussed here can be summarized in the following simplified diagram.

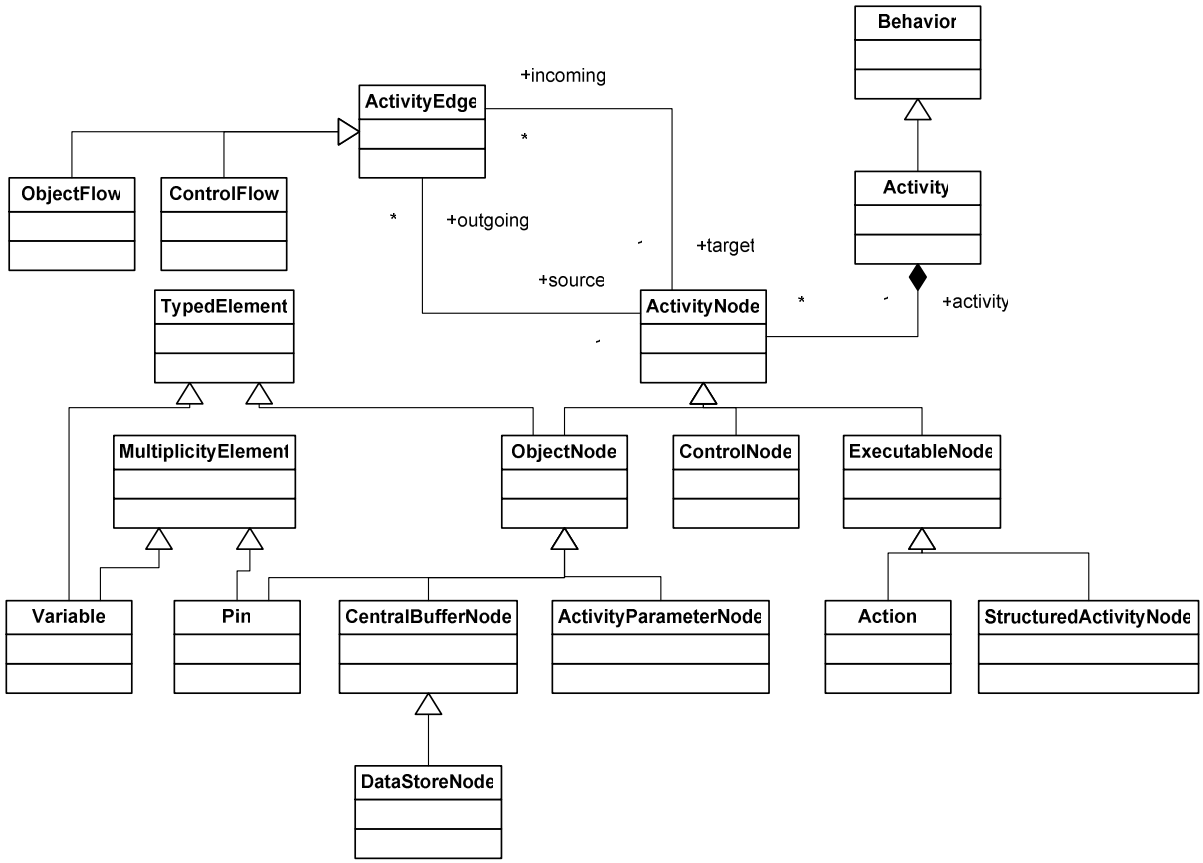


Figure 7 Simplified Meta-classes related with UML Activities

Selection and transformation behaviours can be applied on edges coming out of data store nodes to retrieve information from the store, as if a query were being performed. The ObjectFlow element from CompleteActivities is described in the specification as follows (Object Management Group 2004):
"If a transformation behaviour is specified, then each token offered to the edge is passed to the behaviour, and the output of the behaviour is given to the target node for consideration instead of the token that was input to the transformation behaviour. Because the behaviour is used while offering tokens to the target node, it may be run many times on the same token before the token is accepted by the target node. This means the behaviour cannot have side effects. It may not modify objects, but it may for example, navigate from one object to another, get an attribute value from an object, or replace a data value with another. Transformation behaviours with an output parameter with multiplicity greater than 1 may replace one token with many."

7.2.1.6 Structured Activity Model

There are kinds of activity nodes called *structured activity nodes*, with specializations for sequencing, conditionals, loops, and expansion regions for operating on collections.
That part of the UML Activity model seems especially relevant for UML compliant, platform-independent database application specifications. This is because, in contrast to activity models based on flows, it closely follows the style of typical programming languages.

7.2.1.6.1.1 StructuredActivityNode

This is a kind of ExecutableNode. It represents a structured portion of the activity that is not shared with any other structured node, except for nesting.

7.2.1.6.1.2 ExpansionRegion

This is a kind of StructuredActivityNode. Executes multiple times corresponding to elements of an input collection. It may be described as frame for a *foreach* collection processing instruction known from several programming languages. Because of its collection processing capability, ExpansionRegion may be frequently applicable do describing data processing.

7.2.1.6.1.3 LoopNode

A kind of StructuredActivityNode. Covers the functionality of iterative instructions like *for*, *while* and *do..while*. It allows to define the use of variables during the loop setup and iteration. It may return output as a whole and this way serve as a part of data flow-based behaviour specification.

7.2.1.6.1.4 ControlFlow

It represents an activity edge enforcing the sequencing of activities without resorting to data flow dependency.

7.2.1.6.1.5 ConditionalNode

This denotes a multiple-choice decision element – serves as a structured counterpart of the flow-based DecisionNode.
This class is very universal, however, for the majority of applications, seems to be overly complex. It refers to a condition and a result, the latter connected via output pin to the condition. Constructing a concrete syntax for a construct which supports a result which is not equivalent to the evaluation of the condition is extremely difficult and of no use for a VIDE user. VIDE should thus use a different meta-model which assumes that the result upon which the branching is decided is always equal to the evaluation of the condition. However this simplified meta-model can always be covered by the UML meta-model, such that there is always a transformation into the UML meta-model.

7.2.1.7 Conclusions on the UML Actions and Activities

Expressing the VIDE code in terms of standardised UML elements (including the details of behaviour model) is of significant importance for the interoperability among tools and for opening the way for applying VIDE for various aspects of UML models (e.g. wherever the standardized UML behaviour elements apply). When accessing the suitability of the current UML specification as a base of the VIDE language, two main concerns need to be resolved:

- Selection of the UML subset to be covered by the VIDE language.
- Suitability of existing notions from the data intense applications point of view.

The former issue raises the questions on the direct support of some complex UML notions: e.g. whether the non-binary associations, association classes or qualified associations should be covered with VIDE. Another problem is the absence of the standardized set of operators and functions (e.g. mathematical) in the UML.

The latter issue reveals the need for seamless integration into the UML Activities of the higher level operators known from query languages. While the OCL (Object Constraint Language) (Object Management Group 2003) could be proposed for this purpose, this application of the OCL is currently not adequately supported in the UML meta-model. More detailed discussion of this topic is included in Section 7.6.1.

REQ – Lang 2	Simplified UML meta-model	**MAY**
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 • elements are missing which are located in another needed language (like OCL) it may be changed. A model transformation from the modified meta-model to the unmodified one must always be possible. This weakens REQ – Lang 4.		

7.2.1.8 Requirements for UML-based Action Language

7.2.1.8.1 The Concrete Syntax(es) of the VIDE Language

The main focus of VIDE is to design concrete syntaxes (two or more), both graphical and textual (and possible mixtures), which are well suited to the specific user groups which are addressed by VIDE. Designing the VIDE language well, necessitates a careful collection of requirements on the language.

REQ – Lang 3	User Language & Concepts	**SHOULD**
The VIDE language and VIDE tools presented to a certain user groups SHOULD employ the language that is understood by the user group.		
Description: The requirement refers to the generic requirement REQ – NonFunc 6 but focuses on specific languages that are known to the user groups. For example the use of business language for user group Business Analyst. As a further example, flow models are well understood by these users; concepts known from these models should be presented to the users in a similar way.		

REQ – Lang 4	Compliance with Standards	**SHOULD**
The VIDE language should be compliant with existing modelling standards.		
Description: VIDE should not compete with existing adopted modelling standards, especially those adopted by the OMG, such as UML or BPMN. The use of UML (REQ – User 2) is a particular instance thereof.		

REQ – Lang 5	Deviation from Standards	**MAY**
Deviating from REQ – Lang 4 and according to REQ – User 1, 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.		

Description: Domain specific languages can make life easier for users which want to continue to model using their own business language. In conjunction with REQ – Lang 5, this requirement allows to replace standard-conformant parts with special non-standard domain specific languages which seem more appropriate to ensure suitability for the respective VIDE user groups.

REQ – Lang 6	Modularisation and extensibility	**SHOULD**
The VIDE language should be modular and extensible.		
Description: 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.		

Finally it is important to consider user needs as already described in REQ – User 1:

REQ – User 1	Flexibility and interoperability of VIDE language and tools	**SHOULD**
The VIDE language and tools SHOULD be flexible and interoperable with existing tools		
Description: Industrial development projects utilize different programming languages, frameworks and development tools. Usually users have a complex tool ecosystem at work. It is unrealistic to make VIDE replace existing tools. VIDE should thus smoothly integrate with other tools, i.e., read data produced by other tools and provide data readable by other tools. The VIDE language and tools SHOULD be flexible, interoperable and integrated in the programming languages, frameworks and development tools currently used in industrial development projects.		

7.2.1.8.2 Gap Analysis: Where the OMG modelling landscape needs VIDE

The declared purpose of VIDE, as described in the Requirements (REQ – Lang 4), is to comply with existing standards, especially in the context of OMG's modelling standards. This entails that the VIDE language on the PIM level ('action language') should not aim to replace or modify existing OMG standards.
The following table reveals where the gaps of existing standardised concrete syntaxes are.

	Visual Notation	Textual Notation
Structural / Type Definition	UML Structure Diagrams (Vocabulary of Business Rules)	N/A
Control Flow	BPMN UML Activity Diagrams	**Gap: Potential for VIDE**
State Changes / Data Flow	**Gap: Potential for VIDE**	**Gap: Potential for VIDE**
Side-effect free Expressions	**Gap: Potential for VIDE**	OCL

/ Queries		

This analysis has direct consequences for the design of the VIDE language on PIM level. To name just a few:

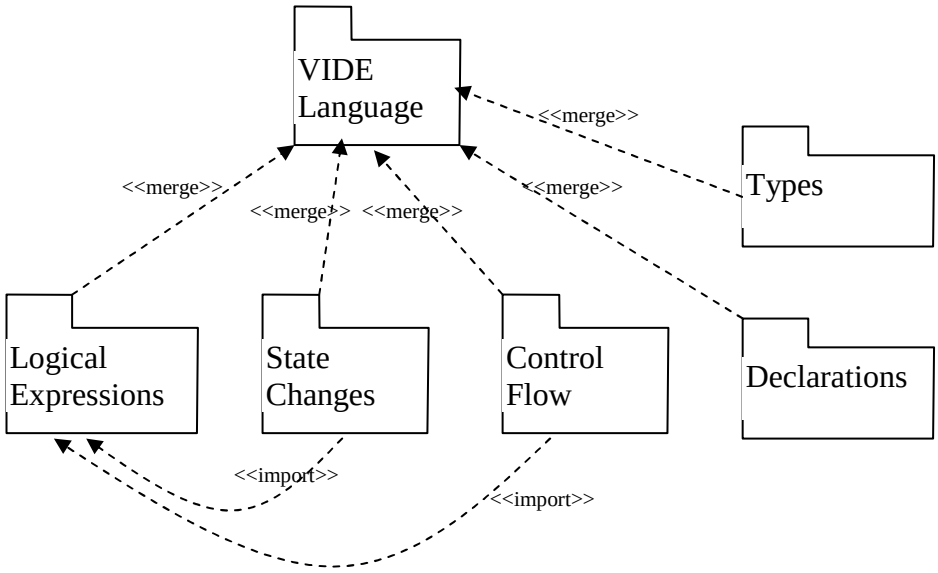
1. Representation of if-then-else. This control flow construct is already present in UML activity diagrams or BPMN and represented by a fork node with outgoing arrows to the different branches. VIDE should (as required by REQ – Lang 4) follow this notation for compliance and for not confusing users which are used to these tried and true notations for quite a long time.
2. Representation of state changes. Though there is no agreed graphical standard here, one could borrow notations from instance diagrams, which depict snapshots of the running modelled system.
3. OCL is a good candidate for the subset of the VIDE language dealing with side-effect free expressions.

7.2.1.8.3 Language Architecture

Pragmatically (see REQ – Lang 5), compliance to standards is not necessarily always the best way to go[6]: Users might be happier with a dedicated domain specific language which suits their needs better than a general, though profilable, language as UML. Fortunately, VIDE's modelling infrastructure is highly adaptable by offering two means of adaptation:

1. Changes to the mapping between graphical/textual elements and the abstract syntax; this allows for slight changes, such as to denote conditional nodes by nested boxes instead of by boxes with connecting arrows.
2. Changes to the meta-model ("abstract syntax") and transformations from an instance of the old meta-model to an instance of the modified one; this allows for heavy-weight changes such as replacing the OCL part of the VIDE language with a different language for expressing logical conditions.

Replacing parts of the VIDE language with new ones would be leveraged by modularising the VIDE language according to a scheme similar to the following:



[6] on the M2 level. Compliance to the agreed standard on M3 level is of course always mandatory.

A possible realisation of the LogicalExpression package would be OCL, another one could represent Java Boolean expressions. Exchanging these two realisations could be supported by the VIDE tool, dealing with the exception that the latter expression language cannot cover all the details of the other.

7.2.1.8.4 Bottom-up Analysis of Necessary Language Constructs

Here we describe typical pattern that can be found in and are typical for business applications on the implementation / business object level. Each pattern is classified, described and an example for an implementation of the pattern is given.
Note: The section focuses on behavioural pattern specific for business applications. Generic behavioural pattern (i.e. loops, cases…) are needed, but not explicitly listed here.

7.2.1.8.4.1 Requirement – "Initialize from constant" Pattern 7

Priority: High
Class: Initialization
Initialize an attribute from the constant. A check is performed if an attribute has been initialized before. If that is the case (the attribute is not in its initial state) the attribute remains as is, if not the attribute is set to a constant value or a list of constants (i.e. Consistent, Inconsistent).
Sample code:
ABAP:

```
  IF me_status IS INITIAL.
   me_status = 'open'.
  ENDIF.
```

JAVA:

```
  if (this.status == null)
      this.status = Status.OPEN;
```

7.2.1.8.4.2 Requirement – Initialize from customizing Pattern

Priority: High
Class: Initialization
Initialize an attribute using customisation. A check is performed if an attribute has been initialized before. If that is the case (the attribute is not in its initial state) the attribute remains as is, if not the attribute is set to a value that is dependent on the specific instance of the system. The customizing value is directly retrieved from a system configuration (usually a customizing table) or customizing method is called that calculates the customizing value.
Sample code:
ABAP:

```
  IF me_status IS INITIAL.
   me_status = "intial_status".
  ENDIF.
```

JAVA:

```
  if (this.status == null)
      this.status =
customzing.getValueFromTable("intial_status");
```

7.2.1.8.4.3 Requirement – Static Consistency Pattern

Priority: High
Class: Consistency

Static rules that state which attributes/relations are (under all circumstances) mandatory, static checks (i.e. type checks) and value checks (i.e. Start Date is smaller or equal to End Date)

Sample code:

ABAP:
```
  IF startDate < endDate.
   ...
  ENDIF.
```
JAVA:
```
  if (startDate < endDate)
     ... ;
```

7.2.1.8.4.4 Requirement – Dynamic Consistency Pattern

Priority: High
Class: Consistency
Consistency checks depending on the current state. Example: “The status of an opportunity is “pending” if it is nether “Lost” nor “Won” or all currencies in a sales document must be the same,

Sample code:

ABAP:
```
  IF opportunity_result == “won” OR opportunity_result ==
“lost”.
   opportunity_status = “pending”
  ENDIF.
```
JAVA:
```
  if (opportunity.result == opportunity.won ||
opportunity.result == opportunity.lost)
   opportunity.status = opportunity.pending;
```

7.2.1.8.4.5 Requirement – Suggestion Pattern

Priority: Low
Note: There is uncertainty whether this behaviour that should be modelled as an isolated behavioural pattern or if the behaviour of suggestions should be modelled using ‘normal’ behaviour models.
Class: User Aid
Suggestions for data entries (on UI level) are quite important to improve the usability of applications. The content of the suggestions comes form various sources for example master data. Examples of such suggestions are

- Responsible employee determination
- Determination of a sales unit party
- Document currency

7.2.1.8.4.6 Requirement – Arithmetic Pattern

Priority: High
Class: Arithmetic
Calculation of an arithmetic value, such as sales forecast, the sum of sales items, discount etc.

Sample code:

ABAP:
```
     me_weightedValue = me_value * me_weight
```
JAVA:

```
this.weightedValue = this.value * this.weight
```

7.2.1.8.4.7 Requirement – Read data from Database

Priority: High
Class: Persistence

Read data from database into memory (table/structure).
Note: Usually this behaviour is supported by the target language or the supporting environment to ease the development of database handling. ABAP for instance has special functions that allow coping database tables into memory. Java supports similar functions with frameworks such as Hibernate [7]

Sample code:

ABAP:

```
READ TABLE table INDEX 1 INTO field
```

JAVA:

```
Connection con;
// ...
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM table WHERE
Index=1");
field = rs.next();
```

7.2.1.8.4.8 Requirement – Write data to Database

Priority: High
Class: Persistence
Write data from memory (table/structure) into database.
Note: Usually this behaviour is supported by the target language or the supporting environment to ease the development of database handling. ABAP for instance has special functions that allow coping database tables into memory. Java supports similar functions with frameworks such as Hibernate. Therefore the requirement is not important for PIM level models.

Sample code:

ABAP:

```
INSERT INTO table VALUES (‘’)
```

JAVA:

```
Connection con;
// ...
Statement stmt = con.createStatement();
stmt.executeQuery("INSERT INTO table VALUES (‘’)”);
```

7.2.1.8.4.9 Requirement – Modify Database

Priority: High
Class: Persistence
Database manipulation
Note: Usually this behaviour is supported by the target language or the supporting environment to ease the development of database handling. ABAP for instance has special functions that allow coping database tables into memory. Java supports similar functions with

[7] http://www.hibernate.org/

frameworks such as Hibernate. Therefore the requirement is not important for PIM level models.

Sample code:

ABAP:

```
     UPDATE table SET field=”updated”.
```

JAVA:

```
     Connection con;
     // ...
     Statement stmt = con.createStatement();
     stmt.executeQuery("UPDATE table SET field=\”updated\””);
```

7.2.1.8.4.10 Requirement – Function calls and declaration

Priority: Medium (considered as covered by UML anyway, however the CHANGING declarations may not be covered)

Class: Function calls

Method calls with parameters (call parameters, return parameters and change parameters). The example below shows only the method calls not the declaration. Most important is that is should be possible to define multiple return parameters.

Sample code:

ABAP:

```
     DATA:
     local_data1 TYPE String.
     local_data2 TYPE String.

   CALL METHOD methodName
    EXPORTING
     param1 = ‘param1’
     param2 = ‘param2’
    CHANGING
     local_method_data1 = local_data1
     local_method_data2 = local_data2
    RECEIVING
     local_method_data1 = return1
     local_method_data1 = return2.
```

JAVA:

```
   //Struct return = {return1, return2, change1, change2};
     String local_data1, local_data2;

     return = methodName(param1, param2);

     local_data1 = return.change1;
     local_data2 = return.change2;
```

7.2.1.8.4.11 Requirement – Exception handling

Priority: High

Class: Exception/Error Handling

Exception/Error handling deals of unexpected program behaviour.

Sample code:

ABAP:

```
  TRY.
      ...
```

```
   CATCH e .
     …
  ENDTRY.
JAVA:
     try {
      …
     }
     catch (Exception e)
          { … }
```

7.2.1.8.5 Language Requirements from the Aspect Oriented Modelling Viewpoint

For a UML model to be executable, its specification needs to be more complete than what can be specified through the standard structural and behavioural models. The actual operations need to be specified.
The UML Action Semantics provides the means to precisely define an operation's behaviour. Based on the UML meta-model, the model elements can be navigated and manipulated in an object-oriented way. Mathematical expressions can be modelled as well as conditional logic. 'Signals' and explicit operation invocations are the means to implement communication between different entities.
A specification this complete, makes it possible to execute it through an interpreter or to generate executable code from it.
In order to provide support for modelling "crosscutting concerns" in a modular way, an extension shall be developed, that supports aspect-oriented techniques in UML models.
Specifically, the most important aspect-oriented concept, the "pointcut", needs to be supported. It must be possible to externally define triggers in model elements or operations, that intercept the "control flow" and lead to execution of an aspect module's operation (advice).
For this, the targetable elements in the ASL (i.e. join point shadows) need to be determined. Aside from model elements (e.g. classes, operations), this could be elements of the ASL itself, like if-statements, loops, function definitions or statements of the ASL, like find, find-all, etc. All keywords of the ASL could potentially be a target (Wilkie, I. and King, A. et al. 2001).
All those targets need to be unambiguously referenceable. Furthermore, the join point context that will be exposed to aspects needs to be defined.
In addition it needs to be investigated, how the aspect weaving can be implemented and integrated into the VIDE toolset. At which level the weaving will be performed is also a subject of research. It is not clear yet, whether aspects will be visible at PSM level at all.
One possible solution could be to perform a model transformation before execution or code generation. Aspect invocations could be inserted at the join points that are selected by the pointcut definitions.
Another possibility would be to make the tools themselves aspect-aware, in a way that they would interpret the pointcut definitions and provide the crosscutting behaviour. This way however, every tool operating on the model would need to be made aspect-aware.

7.2.2 Enterprise Modelling Languages

This section gives a short overview on languages on CIM level. The task for languages on CIM level is to represent the user requirements and give an abstract, high level view on the software. Additionally they should contain model elements that can be easily transformed into (parts of) platform independent models. Therefore five modelling languages are described in

this section. A definite decision of which of these languages VIDE applies will be postponed to the corresponding task in Work Package 7.

Enterprise Modelling (EM) can be defined as the art of 'externalising' enterprise knowledge, i.e. representing the enterprise in terms of its organisation and operations (e.g. processes, behaviour, activities, information, objects, and material flows, resources and organisation units, system infrastructure and architecture). The goal is to make explicit the facts and knowledge that add value to the enterprise or can be shared by business applications and users in order to improve the performance of the enterprise.
Enterprise Modelling Languages (EMLs) should allow building the model of an enterprise according to various points of view such as: function, process decision, economics, etc. in an integrated way.

7.2.2.1 Petri Nets

The Petri Nets are a formal, graphical, executable technique for the specification and analysis of concurrent, discrete-event dynamic systems; a technique undergoing standardization, initially developed by C. A. Petri (Petri 1962) for the specification of concurrent (parallel) systems. The technique has been refined into several subtechniques, such as coloured Petri Nets (Jensen 1997). The main application domains in Enterprise Modelling (process analysis and implementation):

- Manufacturing/Logistics
- Workflow Management

Other Application Domains (behaviour modelling of software products):

- Distributed systems
- Embedded systems
- Hardware and software architectures
- Software engineering in general
- User interfaces

The recognised benefits of Petri Nets in the context of Enterprise Modelling are:

- Modelling power (resource sharing, conflicts, mutual exclusion, concurrency, non-determinism, visual modelling)
- Analysis (deadlock detection, bottleneck analysis, animation, simulation)
- Code generation for Controlling Manufacturing Systems

However, below there is a list of recognised problems with Petri Nets:

- Net size in real world problems
- Difficult to understand due to its operational semantics and to the lack of real abstraction mechanisms
- Code generation in generic distributed systems is hard because many important information (e.g., lack of abstraction) may be hidden or not represented, generating an inefficient code.

It seems, in any field of application, that Petri Nets are positioned at too low level for direct modelling: therefore, they should be used in the context of complete methodologies that, first, provide high level models and then make it possible to transform (automatically or not) these models into (a kind of) Petri Nets.

7.2.2.2 IDEF

IDEF[8] (for Integrated Definition) is a group of modelling methods that can be used to describe operations in an enterprise. IDEF was created by the United States Air Force and is now being developed by Knowledge Based Systems. Originally developed for the manufacturing environment,
IDEF methods have been adapted for wider use and for software development in general.
Sixteen methods, from IDEF0 to IDEF14 (and including IDEF1X), are each designed to capture a particular type of information through modelling processes. IDEF methods are used to create graphical representations of various systems, analysis the model, create a model of a desired version of the system, and to aid in the transition from one to the other. IDEF is sometimes used along with gap analysis.

7.2.2.3 XPDL

XPDL (The Workflow Management Coalition Specification 2005) was developed by the workflow Management Coalition (WfMC). The language uses an XML-based syntax, specified by an XML schema. The main elements of the language are: Package, Application, Workflow- Process, Activity, Transition, Participant, DataField, and DataType. The Package element is the container holding the other elements. The Application element is used to specify the applications/tools invoked by the workflow processes defined in a package. The element WorkflowProcess is used to define workflow processes or parts of workflow processes. A Pattern and XPDL 4 WorkflowProcess is composed of elements of type Activity and Transition. The Activity element is the basic building block of a workflow process definition. Elements of type Activity are connected through elements of type Transition. There are three types of activities: Route, Implementation, and BlockActivity. Activities of type Route are dummy activities just used for routing purposes. Activities of type BlockActivity are used to execute sets of smaller activities. Element ActivitySet refers to a self contained set of activities and transitions. A BlockActivity executes such an ActivitySet. Activities of type Implementation are steps in the process which are implemented by manual procedures, implemented by one of more applications, or implemented by another workflow process. The Participant element is used to specify the participants in the workflow, i.e., the entities that can execute work. There are six types of participants: ResourceSet, Resource, Role, OrganisationalUnit, Human, and System. Elements of type DataField and DataType are used to specify workflow relevant data. Data is used to make decisions or to refer to data outside of the workflow, and is passed between activities and subflows.

7.2.2.4 BPML

The Business Process Modelling Language (BPML) was developed by the Business Process Management Initiative (BPMI). It is based on XML and allows the description of business processes. A process is viewed as a series of activities, a single activity represents a component that performs a specific function. A couple of activities can be composed into more complex activities which are executed within a context which is transmitted from parent to child and allows two activities to share properties. The data flow of BPML is composed by these properties.
The control flow is modelled as ‘activity’, it uses a block-structured approach, wherein each block can itself be considered an activity. Blocks can be nested to arbitrary levels. A process can also be ‘spawned’ or ‘called’. The ‘called’ mode seems to be synchronous as the activity

[8] http://www.idef.com/

is a simple activity. In particular that limits the ability of the 'sub-process process' to 'collaborate' with the parent instance. Spawned processes can be joined via the 'join' activity. As graphical representation of BPML the Business Process Modelling Notation (BPMN) can be used.

7.2.3 Business Process Modelling Notation

The Business Process Modelling Notation (BPMN) is, as UML, an OMG standard (Object Management Group 2006). There is however no official alignment with UML: BPMN is supposed to provide a process oriented view on a system, while UML is strictly following an object-oriented approach. BPMN tries to provide a notation that is understandable by all business users (White 2004), including business analysts (creating the initial drafts of the processes), the technical developers (responsible for implementing the technology that will perform those processes), and the business people (who will manage and monitor those processes).
BPMN defines business process diagram models, similar to flowcharts, consisting of activities as nodes and directed edges describing the flow among these activities. Figure 8 describes a simple business process.

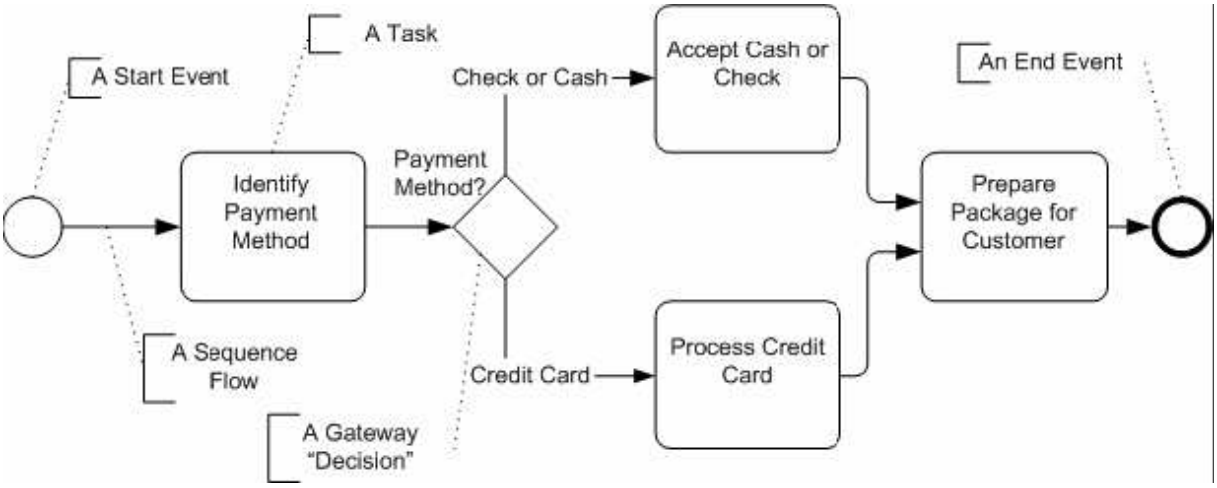


Figure 8: A simple process described with BPMN (White 2004)

BPMN models can be structured within pools and swim lanes and are modular by allowing for collapsed/expanded activities. There is a standardised mapping to BPEL, the de-facto standard for process execution and a number of implementations of BPMN[9]. For VIDE, BPMN could be a suitable modelling standard on the CIM level.

7.2.3.1 Event Driven Process Chain (EPC)

The Event-driven Process Chain (EPC) (Keller, Nüttgens et al. 1992; Scheer 1999; Keller 2000) was developed in 1992 at the Institute for Information Systems in Saarbruecken in cooperation with SAP AG. EPC-models are central elements of BPM, not least because of its use in the SAP R/3 reference model of SAP AG and the ARIS Toolset of IDS Scheer AG. Enterprises model their process data as EPC-models in order to plan, design, simulate and control private enterprise processes. The EPC is a core part of the ARIS-framework and

[9] http://www.bpmn.org/BPMN_Supporters.htm

combines the different views towards the description of enterprises and information systems in the control view on the conceptual level.

An EPC describes processes by the use of alternating functions and events as time-referring state changes Events and functions are linked by the control flow as directional edges. Functions describe activities and events passive states; control flow functions and events can only be connected to each other. To split and join the control flow operator with the occurrences OR, XOR, and AND can be used after functions and events (excepting that the OR- and XOR-Operators must not be used after events). The last remaining element, that could be connected via the control flow are process interfaces. These can be applied at the end and the beginning of an EPC to connect two EPCs from different models. Additionally resources (such as organisational units) can be attached to functions.

7.3 Meta-Modelling Standards

Models are central artefacts of MDSD based projects. A model is described in terms of a meta-model, which represents the vocabulary of the modelling domain. A meta-model conforms to the meta-meta-model, which is abstract enough to allow for meta-model authoring in envisioned development domains. For the rest of this chapter, these modelling levels will be referred to as M1 through M3 correspondingly. This terminology has been proposed by the MOF specification (Object Management Group 2002).

7.3.1 Requirements

The VIDE project poses certain requirements on scalability and extensibility of the underlying modelling infrastructure. These requirements are not of engineering nature only, but are also stressing the conceptual fundament of the modelling infrastructure. The concepts of the modelling infrastructure are cemented in the M3 model, providing the language for M2 model creation.
The M3 model is, under normal conditions, not changeable within the functional scope of the modelling infrastructure. Therefore, its choice should be made with most possible care for details, because of the influence of this decision on the flexibility of the tools, eventually delivered by the project.
Another requirement is the industrial adoption. Because of the limited scope of the VIDE project, it will not be possible, to construct all needed tools in the timeframe of this project. Therefore, the tools landscape should be taken into consideration during modelling infrastructure selection.

7.3.1.1 Scalability requirements

REQ – NonFunc 9	Scalability of proposed solution.	**MUST**
The proposed solution must at least conceptually scale to enterprise level.		

The scalability of a modelling infrastructure, is given by factors like response time of the queries, in face of increasing number of model elements. Because many operations, like transformations, require multiple lookups (e.g. source and target elements with distinctive properties), sub-linear lookup times for elements are required.

The scalability problem has mainly two dimensions. Firstly, the performance with respect to an increasing number of M1 model elements is to be considered. Secondly, the impact of the number of M2 elements on the overall performance of the modelling infrastructure is to be considered. Both problems are tightly coupled. Consider an M2 model with two classes A and B. Further, there exist a number of instances of each class. The first question is how many operations it would take, to deliver all instances of A. Is a complete model lookup required, or is there a notion of extents, which would speed up the lookup? The second dimension, is represented by the question, how the addition of a third class C (not necessarily having any instances) would impact the overall performance. Since the VIDE project is mainly concerned with enterprise scale systems, large numbers of M2 and M1 instances are to be expected. Experience shows, that to model an enterprise, tens of M2 models and thousands of M1 models, having tens of elements each, are required. Linear, or over linear model lookups are not feasible in this environment.

The solution of the presented problems can be achieved through extensive model partitioning. Therefore, the modelling infrastructure, used by the VIDE project can not be based on the closed world assumption. Thus, some notion of absent model elements is needed in the M3 model.

7.3.1.2 Extensibility requirements

The extensibility is defined by the ability of a modelling infrastructure, to allow for the addition of new concepts, without the need to modify the existing M2 models. Like scalability requirements, the fulfilment of extensibility requirements on a modelling infrastructure is rooted in the M3 model.

The most widely accepted convention for model extension is the M2 model federation. Here, a number of M2 models are linked (federated), to allow for richer information sets. The most basic example of M2 model federation is the creation of a diagram of a part of the meta-model population. The population of the domain M2 model will have to be linked with the population of the graphical M2 model. This is needed in order to retain the consistency of the diagram in the face of domain model changes. However, it is very desirable to introduce these links without altering either the domain or the graphical model. The M3 model chosen for the VIDE project will have to provide such linking capabilities.

7.3.1.3 Industrial adoption

REQ – Tool 1	Usage of Industrially Adopted Tools	**MUST**
VIDE must use industrially adopted meta-modelling standards where applicable.		
Description: As decided by partners and as derivation from REQ – Lang 5.		

The industrial adoption is an important factor, because of the lock-in effect of the particular modelling infrastructure. Provided that a modelling infrastructure (and its M3 model) are widely adopted, a large number of tools based on it can be reused without the need to develop them again for the VIDE project.
MDSD projects require an unusual amount of tools, due to the large number of artifacts required by model driven development. These tools can be editors for the particular M3 and M2 model populations, code generators, workflow managers or any other kind of development assisting tools.

Further, the dissemination efforts can largely benefit through the adoption of a widely spread M3 model and the corresponding modelling infrastructure.

7.3.2 Existing M3 Models

A number of M3 Models exist either as a formally defined standard or as an industrial de-facto standard. The two most known M3 Models are the Meta Object Facility (Object Management Group 2002; Object Management Group 2004) (MOF) provided by the Object Management Group and Ecore, provided by the Eclipse Foundation as the basis of the EMF framework (Budinsky, Steinberg et al. 2004).

7.3.2.1 Meta Object Facility (MOF)

The Meta Object Facility (MOF) is an M3 model, proposed by the OMG. There are two relevant versions of this standard, MOF 1.4 (Object Management Group 2002) and MOF 2.0 (Object Management Group 2004). MOF 1.4 is an established standard, with at least one compliant repository (NetBeans MDR (Sun Microsystems 2004)) available. MOF 2.0 is the newer MOF revision, split in two parts: EMOF and CMOF. EMOF is Essential MOF providing the subset of Complete MOF (CMOF). EMOF is conceptually related to the Ecore M3 Model described below. An automatic migration path is being provided, for MOF 1.4 to CMOF 2.0 migration. However, there are no MOF 2.0 compliant production grade repositories available at the moment.

7.3.2.2 Ecore

Ecore is the foundation of the EMF framework. As described above, Ecore is closely related to the EMOF specification. As part of EMF, Ecore is delivered to production environment within Eclipse distribution (as of Callisto release). Therefore, a large user base is ready for EMF based applications. Further, there exist a large number of tools, contributed by the community ready for usage with the EMF Framework.

7.3.3 Feature Comparison of M3 Models

Following table provides an overview of the identified features and their availability in corresponding M3 Models.

Feature	MOF 1.4	CMOF 2.0	EMOF 2.0	Ecore
Extents capability	X	X		
Associations	X	X		
Industrial Adoption				X
Repository Available	X			X

7.3.4 Selection

The partners decided to use EMF as VIDE's modelling framework. The rationale for this decision is the availability of tools for EMF. However, concepts from MOF like Associations should be simulated by means available in EMF.

REQ – Tool 2	Meta-modelling Framework	**MUST**
VIDE must use EMF as it's modelling framework		
Description: As decided by partners.		

REQ – Tool 3	Meta-modelling Concepts	**SHOULD**
VIDE meta-models should be constructed to be compatible with MOF concepts		
Description: As described in this chapter.		

7.3.5 Selected Standard

The consortium selected the Eclipse Modeling Framework (Budinsky, Steinberg et al. 2004) to be the used modelling infrastcructure. EMF is based on the Ecore Metamodel, which is conceptually comparable to EMOF 2.0.

7.4 Model Transformations

Model transformations are needed, to reduce duplication of effort during the creation of correlated modelling artefacts. There two distinct kinds of transformations: Model-to-Model (M2M) and Model-to-Text (M2T). Model-to-Model transformations are normally executed on a model graph, in order to create another, correlated, model graph. Model-to-Text transformations are executed, to create a textual artefact from information contained in a graph based model. There are approaches, to describe Model-to-Text transformations in a Model-to-Model manner (e.g. considering AST of the target artefact as a model graph). These approaches however are not in scope of the VIDE project, due to the immature state of research in that domain.

7.4.1 Importance for the project

Because VIDE focuses on executable models, model transformations will be essential to the project. It will be the responsibility of the transformations to deliver the code (during design or runtime) which will eventually be executed. Due to the importance of this role, the VIDE project poses some requirements on the model transformations.

7.4.2 Requirements

Main requirement on model transformations is their traceability. Thus, when some artifacts are being created via a transformation, the corresponding elements in the source model should be traceable from the created element.
The traceability requirement is of central importance, to model level debugging. This form of debugging enables the user, to observe the results of model's execution in the same notation, used for model authoring.

7.4.3 Available Model-to-Model standards

The simplest way of creating Model-to-Model transformations is the creation of an imperative program, which inspects an existing source model and creates the target model. An imperative implementation can be created in a most unobtrusive way, by implementing the visitor pattern. This approach requires the least number of third party tools, but lacks any support for traceability and tends to break in the face of M2 Model changes. Another way of supporting Model-to-Model transformations is the usage of an available Model-to-Model transformation standard implementation. The rest of this section concentrates on declarative Model-to-Model transformations, as these are considered to be the most maintainable and fruitful way to support the required functionality.
Object Management Group has recently published the Queries/Views/Transformations (QVT)(Object Management Group 2005) standard. This standard targets transformations and queries on MOF 2.0 models. As of now, there is no complete implementation available.
The Eclipse Foundation hosts the ATLAS Transformation Language (ATL) (ATLAS group and LINA & INRIA Nantes 2006), which is a result of the ModelWare[10] project. This transformation language is closely related to the QVT standard and provides a running implementation. Although hosted by the Eclipse Foundation, the ATL project provides bindings for the MOF 1.4 compliant NetBeans MDR modelling repository. ATL does not provide tracing support out of the box. However, approaches allowing the addition of tracing capability to ATL transformations exist. Another transformation language to be evaluated is Tefkat (Lawley and Steel 2005). Tefkat has been developed at the University of Queensland and is available as SourceForge project.

Following table provides an overview of transformation language features.

Name	MOF 1.4 Support	MOF 2.0 Support	EMF Support	Abstract Syntax	Concrete Syntax	Traceability Support
ATL	X		X	X	X	X
Tefkat	?	?	X	X	X	?

7.4.3.1 QVT

As its name suggests the specification MOF QVT (Query / View / Transformation) is intended to provide technology neutral solutions for querying, transforming and specifying views of MOF-based models. The language is built based on parts of MOF and OCL and combines declarative and imperative styles of programming. Such functionality is of significant importance for model transformations assumed by MDA. The document is currently in the "Final adopted specification" state. Its adoption may further raise the importance of using MOF-compliant meta-models.

REQ – Tool 4	M2M Transformation Technology	**SHOULD**
VIDE should use ATL as it's transformation framework, unless it is proven insufficient		
Description: As decided by partners.		

[10] http://www.modelware-ist.org/

7.4.4 Available Model-to-Text standards

Analogously to Model-to-Model transformations, there are numerous Model-to-Text transformation standards.
First, there is the possibility to execute Model-to-Text transformations in an imperative manner. This possibility can be ruled out for the same reasons as this approach has been ruled out for Model-to-Model transformations.
Second, there is the possibility, to use existing template engines like Jakarta Velocity. These engines decouple the form of the transformation output from the input parameters. However, no modelling specific features are currently supported.
Third, there exists specialized Model-to-Text transformation languages like MOFScript (Oldevik 2006). MOFScript is being standardized by OMG and provides features specific to Model-to-Text domain. An implementation of the MOFScript standard is available from Eclipse Foundation. This implementation however lacks important features like traceability.
The fourth alternative is the usage of XPAND language from openArchitectureWare toolset. This language has been proven in real world projects.

REQ – Tool 5	M2T Transformation Technology	**SHOULD**
VIDE should use XPAND as its M2T transformation language, unless proven insufficient.		
Description: As decided by partners.		

7.4.5 Available Text-to-Model tools

The textual parts of the behaviour specification have to be parsed, to acquire their abstract syntax representation. XText is a framework provided by openArchitectureWare and allowing for automated ANTLR grammar generation. The drawback of the Xtext usage is the similarity of the generated M2 model to textual notation's concrete syntax. This requires a transformation from the generated M2 model into envisioned domain M2 model (in this case UML Action Semantics). A valid alternative to XText's approach is manual creation of grammar definitions, followed by automated parser generation via ANTLR or similar parser generators. In this case, the ASTs provided by the parser will have to be mapped by hand onto Action Semantics M2 model.

REQ – Tool 6	T2M	**SHOULD**
VIDE should use XText framework, unless proven insufficient. An alternative can be parsers generated with ANTLR or LPG.		
Description: As decided by partners.		

7.5 Graphical Modelling Frameworks

A graphical modelling framework is required, to allow for easier graphical tool creation on top of domain models created by the VIDE project. There are two possible approaches to graphical tool creation. Firstly, a graphical tool can be programmed explicitly, using an existing graphical framework like GEF (Aniszczyk 2005). Secondly, a declarative graphical framework like GMF (2006) can be used.

The explicit approach has its advantages, due to the immediate feedback to the tool developer. Thus, the tool creation can precede using same methodologies as a transitional software project. However, this approach tends to break, when the domain models are subject to frequent changes.
The declarative approach calls for a mapping between a domain M2 model and the provided graphical model. This mapping is in case of GMF implemented as a population of the M2 model, called the mapping model. This approach has a steeper learning curve, but is more stable in face of M2 model changes. Further, this approach allows for easier synchronization between domain model populations and the corresponding graphical representations.
At the time of this writing, only one stable graphical modelling framework, the Eclipse Foundation's GMF is available. This implementation is tightly bounded to EMF and therefore the Ecore M3 model.

REQ – Tool 7	Meta-modelling Framework	**SHOULD**
VIDE SHOULD use GMF as it's graphical modelling framework		
Description: As decided by partners.		

7.6 Query and Constraint Language Standards

7.6.1 OCL

The Object Constraint Language (Warmer and Kleppe 1999) – originally a part of the UML standard – with version 2.0 of UML and MOF has been provided as a separate specification aligned with the core of MOF and UML. Within the UML and MOF style of model and meta-model definitions the OCL statements serve as the most precise means of model specification (together with less precise class-model based definitions and natural language description). For that purpose OCL was defined to be able to express constraints for any (first-order) kind of UML elements. At the same time the purpose of this language prohibits its statements from causing any side effects. The side effects for a given unit of behaviour can be specified only declaratively and indirectly through the pre- and post-condition clauses. This allows for the successful expression of any functional behaviour. On the other hand, the declarative way of OCL of modelling is commonly assumed to be less intuitive for users which think in terms of actions. Moreover it is presumably a larger gap to come to executable models from a declarative specification (OCL) than from an imperative specification (Action Language).

OCL's "native UML" standard status and fully specified concrete syntax makes the language a viable option for complete platform independent behaviour modelling needed for MDA (Kleppe, Warmer et al. 2003). With its declarative style and operators dealing with collections, OCL may at least inspire introduction of explicit higher level constructs into UML action semantics or even become a part of the action language.
The specification distinguishes two language levels: supporting the UML-specific elements and the core elements common with MOF. The following applications of OCL within UML language are indicated in the document (Object Management Group 2003):

- querying the model
- invariant specification for classes and types
- invariant specification for stereotypes
- pre- and postcondition specification for operations and methods
- guard description (in state diagrams)

- specification of target (sets) for messages and actions
- specification of constraints on operations
- specification of derivation rules for attributes for any expression over a UML model.

OCL expressions can access *properties* of model elements, that is, attributes, association-ends, as well as side-effect-free methods and operations.
The semantics of OCL (Object Management Group 2003) is specified through a UML-based model of its semantic domain and its mappings with abstract syntax. Moreover a strict semantics definition based on set theory is provided in the specification as a non-normative appendix based on (Richters 2002).

OCL could be closely integrated in the VIDE action language. The action language will necessarily need to incorporate a language for expressing expressions, for conditional expressions for instance. This is however exactly what OCL can do. So OCL could be a subset of the VIDE language as discussed below.

There has been a project for visualizing OCL, too (Kiesner, Taenzer et al. 2002). The results give insights into the difficulties of modelling logical expressions graphically, which any language which deals with logical expressions (as VIDE) has to cope with.

In the following subsections we discuss the role of OCL in model driven approaches and compare a purely OCL based approach to MDA with an approach which includes OCL in the action language.

7.6.1.1 MDA and OCL

Considering the OCL from the point of view of MDA, two fundamentally different OCL-based approaches can be discussed:

- purely constraint-based specification using OCL for invariants, preconditions and postconditions (as presented in (Kleppe, Warmer et al. 2003)),
- imperative specification of operation behaviour expressed with UML Actions, Activities and supported with OCL expressions embedded within an action language (Haustein and Pleumann 2004).

Comparison of those two approaches is illustrated below in two examples based on the same schema consisting of classes *Employee* and *BonusGiven*. Accordingly to the VIDE project aims we focus on the latter approach, but present also the constraint-based solution to show the different purpose of the OCL there and to discuss pros and cons of the imperative approach assumed by VIDE. The two examples has been constructed so as to show the strengths and drawbacks of those two approaches.

7.6.1.1.1 Domain classes used in these examples:

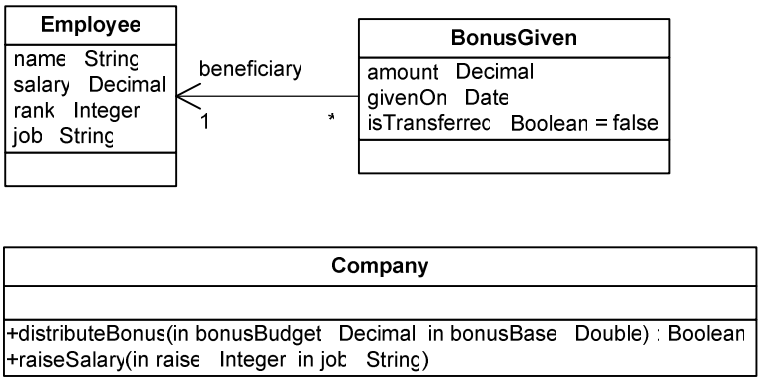


Figure 9 Domain classes for the OCL example

The *rank* attribute reflects the current result of the evaluation of an employee's achievements. The higher it is, the higher the bonus amount should be when it is distributed.

The objects of a class BonusGiven are created to indicate the decision of paying a given employee the bonus of particular amount. When the amount is paid to the employee, the *isTransferred* flag changes to true.

7.6.1.1.2 Example 1

Procedure to specify:

distributeBonus(bonusBudget : **Integer**, bonusBase : **Real**) : **Boolean**;

Natural language description of the intended behaviour

When this method is invoked, an employee should receive a bonus exactly of the amount that is calculated as a product of: (rank * bonusBase) or not receive the bonus at all. The fact of giving a bonus is reflected in the creation of the BonusGiven object connected to appropriate Employee object.
The sum of bonuses to be paid may not exceed the value of bonusBudget.
In case the calculated sum of all bonuses does not exceed the bonusBudget, all employees should receive the bonus calculated as above and the procedure should return true.
Otherwise, the employees should be ordered descending by the value of the rank. Then for each employee in that sequence an attempt should be made to give him / her the bonus. The remaining amount of bonusBudget should be calculated. In the iteration in which the remaining budget becomes not sufficient to give another bonus, the BonusGiven object is not created and the whole procedure returns false.

UML Behaviour + OCL specification using ad-hoc Java-like syntax for actions

Code only:

```
distributeBonus(bonusBudget : integer, bonusBase : real) : boolean{
      moneyRemaining: real = bonusBudget;
      Sequence<Employee> s =
```

```
      new ExpressionInOcl("Employee.allInstances()->sortedBy(rank)");
      for (emp : Employee in s){
            bonusCalculated : real = (emp.rank*bonusBase);
            if(bonusCalculated<=moneyRemaining){
                  moneyRemaining-=bonusCalculated;
                  new BonusGiven(emp, bonusCalculated, Date.today() );
            }
            else return false;
      }
return true;
}
```

Code commented to indicate the underlying action primitives involved:
Note that this syntax is in some constructs of a higher level nature than the respective UML behaviour elements. Not all actions and flows are explained in the comments.

```
distributeBonus(bonusBudget : integer, bonusBase : real) : boolean{
      moneyRemaining: real = bonusBudget;
      // AcceptCallAction:
       // OutputPin: bonusBudget : integer
            // -> Variable: moneyRemaining : integer;
       // OutputPin: bonusBase : real
            // -> Variable: bonusBase : real;
Sequence<Employee> s =
      new ExpressionInOcl( "Employee.allInstances()->sortedBy(-rank)" );
  for (emp : Employee in s){
      //  ExpansionRegion;  ExpansionNode  =  Employee  {ordering  =  FIFO;
      // upperBound = *}
      bonusCalculated : real = (emp.rank*bonusBase);
      if(bonusCalculated<=moneyRemaining){
            moneyRemaining-=bonusCalculated;
            new BonusGiven(emp, bonusCalculated, Date.today() );
      }
      else return false; // ReplyAction
  }
  return true; // ReplyAction
}
```

The ExpressionInOcl is assumed to be connected with the UML 2 meta-model as a meta-class specializing the OpaqueExpression. This solution emphasizes the distinction of these two languages, resembling to some extent the one of SQL embedded into general-purpose programming language code. Since the UML Expression does not specify the multiplicity, it is expected that multivalued results of OCL expressions will be described as OCL collections (i.e. Bag, Set, OrderedSet, Sequence). It is not fully clear, if such results can be considered compatible with UML Behavior Pins and Parameters, but we make such assumption here.

Pragmatically, OCL expression successfully performs its job, reducing the amount of code needed to retrieve and order the Employee objects for the bonus assignment. The details of integration such expressions into UML Behavior elements that require clarification are:

- compliance of OCL collections with UML Pins and Parameters,
- support of UML Actions and Activities for OCL-generated tuples. Considering the data manipulation constructs available, it seems that the tuple should be represented at the UML side as an instance of single Pin or Parameter whose type is a Tuple (which could be a specialization of UML's DataType) rather than a set of Pins or Parameters one for each tuple field.

The above example could be easily extended to illustrate the need for tuple. Rather than being a ready attribute, the rank could be calculated for each employee from some other data (e.g. evaluations). Then, the iteration would work not on the sequence of ***Employee*** *objects, but rather on the sequence of tuples:* ***{e : Employee, rank : real}****.*
Based on the above considerations, there seems to be no reason for not including the notions of tuple and various kinds of collections as the inherent, predefined elements of UML specification.
To fully avoid the "impedance mismatch" between the two languages, the elements describing the OCL expression should be treated as inherent part of UML Behavior (which would make the ExpressionInOcl non-opaque). Alternatively, UML Activities should be accompanied with elements covering this part of OCL which has not yet its counterparts in UML Behavior.

UML + OCL purely constraint-based solution

```
context Company::distributeBonus(bonusBudget : integer, bonusBase : real) : Boolean
let: allEmployees = Employee.allInstances() in

-- not more than necessary left
post: allEmployees->collect(e|e.bonusGiven.amount)->sum()
      +min(allEmployees->select(e|e.bonusGiven.size()=0)->collect(rank*bonusBase))
      > bonusBuget

-- if one employee got no bonus all lower-ranked got it neither
post: allEmployees->forAll(e1,e2 | e1.rank<e2.rank and e1.bonusGiven.size()=0
      implies e2.bonusGiven.size=0)

-- if one employee got a bonus all higher-ranked got it too
post: allEmployees->forAll(e1,e2 | e1.rank<e2.rank and e2.bonusGiven.size()>0
      implies e1.bonusGiven.size>0)

-- budget not exceeded
post: allEmployees->collect(e|e.bonusGiven.amount)->sum()<bonusBudget

-- all employees who got bonus, got the right amount
post: allEmployees->forAll(e|e.bonusGiven->size>0 implies
      e.bonusGiven.size=1 and e.bonusGiven.amount=e.rank*bonusBase)

-- we return true if all employees got the bonus
post: result=allEmployees->forAll(e.bonusGiven.size>0)
```

This example shows that it is not necessarily the case that OCL style specifications are easier to understand. While the OCL specification distributes information to several constraints, the action language style specification is more compact and more closely resembles the informal natural language description. On the other hand, the OCL specification allows for several implementations (one could use recursion instead of looping for instance) and just describes the result, while the action language specification is exact in the algorithmic details. There is another issue worth mentioning: in OCL, it would be easy to make the specified method change something completely irrelevant, such us changing the name of the employee. In action language, this cannot be done, since only that what is specified is executed, nothing more. This problem is referred to as the frame problem in literature, and there are several approaches to cope with it in OCL, but it is naturally easier to deal with it in action language.

7.6.1.1.3 Example 2

Procedure to specify:
raiseSalary(raise : **integer**, job : **string**);

Natural language description of the intended behaviour

The method iterates over all the Employee objects and increases the *salary* attribute in the objects having the attribute *job* equal to the *job* parameter by the amount provided by the *raise* parameter.

UML Behavior + OCL specification using an ad-hoc Java-like syntax for actions
(Not all actions and flows are explained in the comments.)

Code only:

```
raiseSalary(raise : integer, job : string) {
      String query = "Employee.allInstances()->select(job==\""+job+"\")";
      Sequence<Employee> s = new ExpressionInOcl(query);
      for (emp : Employee in s){
            emp.salary = emp.salary + raise;
      }
}
```

Code commented to indicate the underlying action primitives involved:

```
raiseSalary(raise : integer, job : string) {
      // AcceptCallAction:
       // OutputPin: raise : integer
            // -> Variable: raise : integer;
       // OutputPin: job : real
            // -> Variable: job : string;

      // ReadVariableAction: job

String query = "Employee.allInstances()->select(job==\""+job+"\")";
// AddVariableValueAction: result; isReplaceAll = true
Sequence<Employee> s = new ExpressionInOcl(query);
// ReadVariableAction: result;
  for (emp : Employee in s){
      //  ExpansionRegion;  ExpansionNode  =  Employee  {ordering  =  FIFO;
      // upperBound = *}
      emp.salary = emp.salary + raise;
      // ReadVariableAction:              raise
      // ReadStructuralFeatureAction:     salary
      // FunctionBehavior: salary + raise
      // AddStructuralFeatureValueAction: salary; isReplaceAll = true
  }
}
```

This example uses mainly the UML Behavior notions. With the assumptions as in the previous example (that is, the assumed compatibility of OCL and UML types), we may use OCL expression to ease the selection of objects being updated. Note that the resulting style is typical to the query embedding known from popular programming languages used with SQL. The mismatch resulting from such arrangement is not as severe as in case of embedded SQL since the types of data structures being extended are considered fully compliant. Similarly like in case of SQL, the OCL expression is opaque (it is treated as text) and thus raises the question about the possible impedance mismatch in processing of such code.
In fact, the ExpressionInOcl as a specialization of UML OpaqueExpression is not prepared to accept any parameters. We thus assume that the concatenation providing the value of job

parameter is performed within the query variable, which is then passed to ExpressionInOcl as its body attribute. The specification of UML does not state explicitly whether the body is allowed to be dynamically constructed. Another potential drawback of such usage is the inability to (re)use the iteration encapsulated in the OCL expression to perform the updates in the same loop.

UML + OCL purely constraint-based solution

```
context Company::raiseSalary(raise : integer, job : string)

-- Each employee of matching job gets the raise if his/her salary

post: Employee.allInstances()->select(e | e.job = job)
            ->forAll(e | e.salary = e.salary@pre + raise)
```

7.6.1.2 Conclusions

From looking at the simple examples above, we may observe that either of those approaches may turn out to be advantageous for one or the other method, depending on the characteristics of the business requirement.
Using constraints can be also considered the ultimately abstract specification of behaviour, since no accidental algorithmic choices are fixed and the intended result is described. At the same time however, using constraints solely may be problematic for the following reasons:

- The business requirements described imperatively could result in fairly complex set of constraints, making it difficult for an analyst to formulate them and verify their completeness (see the first of our examples).
- The ultimate transformation of requirements into the imperative code (e.g. of the target platform) is much more challenging than in case of imperatively specified behaviour. Consider for instance the easy specification of the square root: `result*result=x` in OCL, this is non-trivial to make executable.
- The constraint approach also requires maintaining a level of granularity of behaviour units sufficiently fine to enforce with pre- and postconditions all the meaningful steps of the business logic having specific requirements connected with it.
- One often explicitly wants to say which method is called from which other method (grey box specifications). OCL allows this in a certain way, but one cannot specify the temporal order of these calls. Nevertheless such grey-box specifications are needed if one wants to describe a model-view-controller pattern for instance.

In the action language approach, the use of embedded OCL expressions is very compelling in order to:

- ease the specification of iterating on the data thanks to the query operators provided by OCL,
- avoid loosing the level of abstraction when coming from the model of behaviour, through UML Actions, and towards the target platform language (possibly supporting queries).

There are some limitations though. OCL is capable of accessing any features of UML models and instances and – in its primary purpose – provides the results of its expressions for UML constraint checking. On the other hand, the integration with UML in the area of exchanging data structures between OCL expressions and UML Actions is an issue. As suggested above, a non-opaque treatment of OCL expressions inside the UML Actions and Activities would be expected to achieve truly seamless integration of those constructs. We argue that while some

heterogeneity may appear at the program code level (e.g. the mentioned SQL embedding), there is no reason to allow for it at the PIM level. Another aspect of the OCL integration is the ability to pass parameters into the OCL expressions (as shown in the example 2). This issue seems to be solvable with the non-opaque treatment of OCL expressions inside UML behaviour models.
The landscape of OCL-related specifications is more complicated though. It is necessary to note that for the purpose of MOF QVT specification (Object Management Group 2005) the OCL has been recently provided with imperative statements (so-called Imperative OCL) and is to a great extent analogous to existing elements of UML Behavior. This may be considered as a kind of redundancy within the OMG specifications and as a sign that the development of UML/OCL and MOF/OCL takes the form of two, separate developmental tracks.
Taking those issues into account, VIDE needs to seek for the solution for integrating the query language capabilities into UML behaviour which would:

- Firstly, guarantee a fully seamless and uniform combination of those language constructs and not to limit their expressive power.
- Secondly, minimize the redundancy between different OMG specifications and minimize the need to introduce new language elements.

REQ – Tool 8	Use of OCL	**SHOULD**
VIDE should re-use existing standards (REQ – Lang 4) as UML (REQ – User 3), and in particular OCL. The goal is to achieve a seamless integration with the concrete syntax of the action language to be developed.		

7.7 Related Standards

7.7.1 XMI

XML Metadata Interchange (Object Management Group 2005) is a MOF-based specification providing the rules of XML serialization of models, allowing their transfer between standard-compliant tools. This specification is generic in this sense, that it is not limited to UML models serialization, but instead may handle any meta-model defined using MOF. This results in both models and meta-models being the subject of exchange: From a MOF defined meta-model a XML Schema definition is produced and the models are expected to match that schema. As such the specification becomes important for exchange of whole models between tools (in contrast, it would be rather clumsy to implement tightly coupled tool integration around XMI).

7.7.2 CWM

Common Warehouse Meta-model is a remarkable example of a standalone domain specific meta-model standardized as a part of the OMG family of modelling specifications. Specification is described as “interfaces that can be used to enable easy interchange of warehouse and business intelligence metadata between warehouse tools, warehouse platforms and warehouse metadata repositories in distributed heterogeneous environments” (Object Management Group 2003).

Since its meta-model is specified independently (and not as a UML profile), this exemplifies the application of MOF-based meta-modelling to construct platform independent modelling means for a particular domain. It is interesting to observe how the concrete notions of particular technology are abstracted into platform neutral modelling constructs.
VIDE language development may encounter the need of similarly in-depth meta-model-based extension of existing modelling infrastructure. The required extensions or customizations may come from two directions. First, bringing new programming solutions onto platform independent modelling level may require additional language constructs. Second (as suggested by the idea of Pervasive Services of MDA) some features of target implementation platform may have enough in common, such that their abstract form should be available during specification of PIMs.

8 Tool Selection

Like many software development approaches, MDA could achieve little without tool support. VIDE aims to enhance the MDA approach by providing toolsets for model driven development whereby application code is generated from CIMs and PIMs. There are numerous tools that already support MDA development by offering varying functionality for MDA specification. To determine the issues that VIDE must address, it is vital that some of the main MDA tools are studied, in order to learn from existing work and to improve on it. The review method adopted is based on (Kitchenham and Jones 1997) and is focussed on a number of features that the OMG considers essential for software development using the MDA approach. Some of these include the capability of tools to support PIM to PSM transformation, and the automated generation of code from one or more PSMs. What follows then, is an overview of industry and research tools that exhibit some of these features.

8.1 MDA Tool review

This section outlines a set of MDA tools and the features they each provide. The summary of the tools is provided in a table, and the features being investigated are provided below. The features are derived from literature on MDA (academic papers and the OMG website).

The tools discussed are OptimalJ (Compuware 2006), ArcStyler (Objects 2006), Constructor (DotNetBuilders 2006), Codagen Architect (Codagen 2006), Objecteering (Objecteering 2006), Ameos (Aonix 2006), Together Architect (Borland 2006), XMF Mosaic (Xactium 2006), Jamda (Boocock 2006), PathMate (PathFinderSolutions 2006), NetBeans (NetBeans 2006), Rational XDE Developer (IBM 2006), Eclipse Modelling Framework (Eclipse 2006) and Websphere (IBM 2006).

The tools

Short word	Full Name	Company
OJ	OptimalJ	Compuware
AS	Arcstyler	Interactive Objects
CT	Constructor	Dot Net Builders
CA	Codagen Architect	Codagen
OG	Objecteering	Objecteering Software [SOFTEAM]
AM	Ameos	Aonix
TA	Together Architect	Borland [Inprise]
XM	XMF Mosaic	Open source
JD	Jamda	Open source
PT	PathMate	IBM
NB	NetBeans	NetBeans & Sun Microsystems
RX	Rational XDE Developer	IBM
EMF	Eclipse Modelling Framework	IBM
WS	Websphere	IBM
Aris	Aris Toolset	IDS Scheer

Features studied are summarised as follows:

- CIM – whether or not the tool provides support (via notations such as use cases, or activity diagrams) for CIM modelling.
- PIM – whether or not a tool provides support for PIM modelling.
- PSM – whether or not a tool provides an explicit support for PSM modelling by further development of PIM or transformation of PIM to PSM.
- UML 2.0 – whether or not the tool supports UML2.0 compliant modelling activities.
- MOF 2.0 – whether or not the tool offers MOF2.0 compliant meta-modelling.
- Action Semantics – whether or not the tool provides an implementation of action semantics for production of executable models.
- UML profiles – whether or not the tool supports the authoring of UML profiles.
- XMI – whether or not the tool supports the XMI standard.
- CWM – whether or not the tool provides support for the CWM standard.
- QVT – whether or not the tool supports the QVT standard via some form of implementation of the standard.
- OCL – whether or not the tool supports expressing constraints over modelled objects using the OCL standard.
- PIM→ PSM→Code – whether or not the tool supports MDSD on all the three phases.
- PIM←PSM←Code – whether or not the tool supports reverse engineering of models in these three phases.
- PIM→Code – whether or not the tool supports direct code generation form PIM models.
- PSM→PSM bridge – whether or not the tool offers a bridge to derive PSMs for different specific platforms (e.g. Java or .NET).
- Legacy code →PSM – whether or not the tool supports production of PSM from existing legacy code.
- Traceability of transformations → whether or not the tool allows for modellers to trace transformations between models (e.g. between CIMs and PIMs or between PIMs and PSMs).
- Transformation based on patterns – whether or not the tool provides patterns for transforming CIMs to PIMs, PIMs to PSMs or PSMs to code. These patterns may be domain specific at lower levels, or platform specific at later levels.
- Merging of models – whether or not a tool provides capability to merge different models (e.g., 2 or more PIMs or 2 or more PSMs).
- More than one implementation platform – whether or not the tool supports derivation of application code for more than one target platform (e.g., C++, Java, or SmallTalk)

Feature	OJ	AS	CT	CA	OG	AM	TA	XM	JD	PT	NB	RX	EMF	WS	Aris
Support for:-															
CIM					✓	✓	✓					✓			✓
PIM	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
PSM	✓		✓	✓	✓			✓	✓		✓	✓	✓*		
UML 2.0	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
MOF 2.0	✓	✓		✓	✓	✓		✓		✓	✓	✓	✓	✓	
Action Semantics						✓	✓			✓			✓	✓	
UML profiles		✓		✓	✓	✓									
XMI	✓	✓		✓	✓	✓	✓	✓	✓		✓	✓	✓		
CWM															
QVT								✓							
OCL								✓					✓		
PIM →PSM→Code transformation	✓		✓		✓				✓		✓	✓	✓*		
PIM← PSM←Code transformation	✓		✓		✓				✓		✓				
PIM→Code transformation	✓	✓	✓	✓	✓	✓	✓	✓		✓			✓	✓	
PSM→PSM bridge					✓		✓								
Legacy code →PSM transformation	✓	✓	✓		✓	✓	✓					✓			
Transformation based on patterns	✓	✓			✓	✓	✓	✓	✓						
Traceability of transformations	✓	✓	✓	✓	✓	✓	✓			✓	✓		✓	✓	
Merging of models	✓				✓					✓			✓	✓	
More than one implementation platforms	✓	✓	✓	✓	✓	✓	✓	✓				✓			

Table 3: Overview of tool support for modelling standards

✓ = Direct support ✓* = Support through 3rd party plug-in

8.1.1 Tool overview

There is an emerging pattern across the tools regarding the modelling notations used. The notations used for CIM construction (where CIM modelling is supported) are UML use case diagrams and activity diagrams, also class diagrams at a conceptual level. PIM modelling is predominantly by way of a UML class diagram, and many tools subsequently base the PSM model on the PIM class diagram in order to reach their target platform. Other observations include:

- Only a few tools (such as NetBeans) describe an elaborate mechanism by which an implementation of MOF is made.
- Other tools such as OptimalJ implement transformations using patterns.
- Most of the studied tools do not indicate how they implement the OMG's MDA standards (e.g., MOF and QVT).
- Most tools have an implementation supporting XMI.
- Some tools support MDA by following recommendations from OMG; others are 'inventive' about how they provide MDA supporting features within their environments.

- Some of the studied tools only support direct generation of code from PIMs, which also means that the construction of PIMs is a process that involves embellishing PIMs with some formal language.

Different approaches have been adopted by the vendors of the tools reviewed. For example, there are MDA tools that adopt a component based approach to providing MDA functions. The Eclipse IDE is one such example, providing an infrastructure for developing Java applications, and a plugin mechanism supporting the MDA development approach based on the Eclipse Modelling Framework (EMF). Models can be specified using annotated Java, XML, or using Rational Rose, then imported into EMF. From a model specification, EMF provides a transformation mechanism for producing a set of Java classes for the model. Finally, EMF provides the foundation for interoperability with other EMF-based tools and applications, such as ATL, and WebSphere.

Other examples include the NetBeans project (which has seen the production of the NetBeans IDE) has provided an implementation of MOF using the Metadata Repository (MDR). The MDR implements the OMG's MOF standard and integrates it into the NetBeans Tools Platform (Matula 2003; NetBeans 2006). Similarly, Objecteering offers support for UML (including model merge), PSM and transformation profiles for Java, C++, C#, CORBA, SQL and even Fortran. The model transformation is specified in a transformation language called J (this is supported through a Java-based API).

In the review of tools conducted thus far, it is important to note that whereas most tools support code generation from PSMs, there are often issues found with the completeness of the code generated. In all the tools presented here, manual ‘tweaks’ of the generated code are required to varying degrees.

8.2 Tools from the European Research Project MODELWARE

MODELWARE is a project co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Program (2002-2006). The overall objective of MODELWARE is to improve productivity in software development. This objective is pursued by contributing to the realization of the vision of model-driven development (MDD). MODELWARE participates in the realization of this vision by pursuing three main goals:

- Improve MDA technologies through new modelling technologies and tools, and through generalized interoperability techniques between tools.
- Define the appropriate software development processes that take advantage of the MDD evolution. MDD maturity levels are defined within this task.
- Use these technologies and processes on large software application developments, in order to measure the ROI and to prove the added value of MDD with the new techniques and processes.

MODELWARE is a large research project (IP: Integrated Project) with 19 partners. It has already increased the knowledge and maturity of its participants. Lessons can already be learned from this project and its experiments.

MODELWARE contributes to realizing the vision of MDD by taking up three main challenges: (1) new development processes must be elaborated, with better and more mature tools to support them; (2) know-how must be reusable across development projects; (3) tool chains must be open to allow interoperability and substitutability between tools, thereby

avoiding tool vendor lock-in and allowing existing models to be capitalized. In the MODELWARE MDD whitepaper[11] , these challenges are described in detail, and the way MODELWARE addresses these challenges is explained.
MODELWARE has produced research results on model transformation, model composition, model weaving and on the MDA Tool Component notion which is now proposed for standardization, and for architecture modelling.
MODELWARE has developed tools dedicated to model driven development, most of them being based on the Eclipse EMF infrastructure. The tools that are the most relevant to the VIDE goals are the following ones:

1. ModelBus[12]: ModelBus enables a transparent integration of model based tool. It is an infrastructure based on the Service Oriented Architecture. The functionalities provided by tools are considered as modelling services. Other tools can benefit from these services using them in a transparent way. The key feature of ModelBus is possibility to exchange models in heterogeneous formats. ModelBus provides an abstraction layer for service description, run-time infrastructure and an integration toolkit for Java-based tools. The Orchestration Tool allows sequencing service execution and in this way provides a means for constructing highly-reusable tool chains. Several commercial tools have been adapted to be connected through ModelBus, the Objecteering Case tool being one of them.
2. MDA Modeler tool set: This tool allows for modelling MDA extensions to the UML, and to package a specific MDA approach into MDA Tool Components. It has been developed on top of the Objecteering case tool. This tool provides reusable services over ModelBus.
3. OCL Tool[13]: An open source OCL2.0 interpreter has been developed. It is based on the Kent OCL library and provides services over ModelBus.
4. Model Transformation Tool Suite (MTTS): This is a group of three model transformation engines: ATLAS[14], QVT from France Telecom and MOFScript[15]. .ATLAS is a general purpose open source model transformation infrastructure, MOFScript is a language and an engine that allows to generate text (such as code typically) from the model. MOFScript is an open source implementation.
5. IBM has developed a tool for model simulation and tests – it offers an interesting approach that could be reused within the VIDE project: the model can be simulated and becomes testable.

The MODELWARE project has defined bespoke software development processes adapted to MDA approaches, and has defined the maturity levels that are relevant for organization using MDA.
The developed approaches and tool suites have been validated using large industrial use. These use cases have produced indication on the ROI that can be expected while using MDA.

The technologies used and developed within the MODELWARE project are of a high relevance for the VIDE project. MDA Tool Components can be used to adapt UML for VIDE specific purposes. The model transformation technologies can be reused within the VIDE project. For example, the code generation that the VIDE project will provide can be based on

[11] MODELWARE MDD Whitepaper (Modelware/I6.1) : http://www.modelware-ist.org/index.php?option=com_remository&Itemid=74&func=download&id=87&chk=f119523ab5e60e59f2fe01b941dd1734
[12] ModelBus white paper: http://www.eclipse.org/mddi/ModelBusWhitePaper_V1.2.pdf
[13] http://oslo-project.berlios.de/
[14] http://www.eclipse.org/gmt/atl/
[15] http://www.eclipse.org/gmt/mofscript/

MOF2text implementations such as MOFScript from MODELWARE. The ATLAS model transformer is one candidate for internal model transformations that might be necessary. The simulation tool from MODELWARE, which is not open source, can provide input on how to use the UML language for modelling more completely the behaviour in an executable way. The ModelBus is an integration mechanism that is worth considering as a possible architectural approach.

8.3 Modelling and Language Implications for Tool Selection

This review has shown that there is a broad range of activities that currently support the development of the MDA/MDD method through the provision of frameworks, languages and tools. In order to reach a conclusion on a particular development tool set (and associated frameworks and standards), it is necessary to consider the modelling requirements as they are set out in Chapter 7.
A brief summary of the VIDE modelling requirements indicate that:

- VIDE should comply with MDA standards (e.g. UML).
- Related standards may be used to extend VIDE (e.g. CWM).
- Interoperability standards should be adopted (e.g. XMI).
- CIM level models should include business modelling concepts.
- Transformation between CIM and PIM models should be supported by VIDE.
- VIDE should be extensible via a plug-in mechanism.
- Action semantics should be used for producing executable models.
- An extension of the action semantics should be provided to enable modelling of aspects.
- Already adopted industrial modelling standards (e.g. UML, Ecore) should be adopted by VIDE.
- VIDE will use EMF as its modelling framework.
- VIDE will conform to MOF and its meta-models will be MOF compliant.
- Model transformation framework within VIDE are planned to be implemented through ATL.
- Model to text transformations will be by XPAND transformation language.
- Text to (diagram) models will be by use of XTEXT framework.
- Graphical modelling will be done using Eclipse's GMF.

The impact of these decisions across the scope of the VIDE project is described in the table below:

	CIM	PIM	PSM
Standards	BPMN UML MOF XMI QVT MDR	UML2.0 – AS UML XMI EMF-Ecore QVT CWM JMI	JMI XMI
Frameworks	Zachman ARIS CIMOSA IDEF ARIS ATL MDR	IDEF EMF NetBeans ATL MDR Xtext GMF	EMF ATL

	Xtext		
Languages	Petri Nets BPML EPC MOF UML CWM XPDL ATLAS MOFScript XPAND OCL	UML2.0-AS EMF-Ecore ATLAS TefKat OCL MOFScript XPAND	EMF-Ecore Java
Tools	ARIS ANTLR	Eclipse ATLTrace NetBeans ANTLR	Eclipse ATLTrace NetBeans

Table 4 Outline of standards, frameworks, languages & tools discussed in section

In the following sections, potential software tools and frameworks from this review (including a few bespoke entities where appropriate) are matched against VIDE modelling demands.

8.3.1 VIDE Development Tool Requirements: Support for Standards

A number of requirements have been identified in this section that relates modelling and imperative language standards to the levels of CIM, PIM and PSM.

REQ – Tool 9	CIM modelling standards	**MAY**
VIDE may support CIM level modelling with BPMN; where there is inadequate or no support for BMPN, VIDE may provide CIM modelling capability with UML activity diagrams.		

Tools available: Appian Enterprise (http://www.appian.com/), and TeamWorks (www.lombardisoftware.com) for BPMN support; Rational Rose, Together, and Objecteering (for activity diagrams).
Selection: Bespoke development.

REQ – Tool 10	PIM, PSM modelling standards	**SHOULD**
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).		

PIM Tools available: Eclipse with EMF; Rational Rose; Eclipse UML (from Omondo).
Selection: Eclipse & EMF
PSM Tools available: Objecteering, OptimalJ, Together, Eclipse/EMF.
Selection: Eclipse framework

8.3.2 VIDE Development Tool Requirements: Support for Modelling Frameworks

The following requirements reflect some of the supporting software APIs/frameworks that have been identified for inclusion within the VIDE development configuration.

REQ – Tool 11	Framework for CIM, PIM, PSM modelling	**SHOULD**
VIDE SHOULD adopt the ATL framework as its transformation framework, and		

should use XPAND for model to text transformations. VIDE SHOULD adopt EMF as its framework for PIM modelling VIDE SHOULD adopt EMF as the meta-modelling framework.

CIM Tools available: ATL Model Weaver, OpenArchitectureWare toolset.
Selection: ATL
PIM Tools available: Eclipse, ATL Model Weaver
Selection: Eclipse framework, ATL
PSM Tools available: Eclipse, Eclipse UML, ATL Model Weaver
Selection: Eclipse framework, ATL

8.3.3 VIDE Development Tool Requirements: Support for Modelling Languages

Here the requirements for specific, executable or automatic transformation formalisms are expressed as requirements for CIM, PIM and PSM levels.

REQ – Lang 7	Language for CIM, PIM, PSM modelling	**SHOULD**
VIDE SHOULD support requirements definition tasks and business process description with BPML VIDE SHOULD adopt action semantics for the modelling of executable PIM models VIDE SHOULD provide support for target PSM environments e.g. Java, C++, or SmallTalk; VIDE should provide platform implementation mappings in PIMs or CIMs.		

CIM Tools available: no known implementations of BPML; may use sister specification – BPMN with possible support from Appian Enterprise or TeamWorks technologies.
Selection: --
PIM Tools available: no tool known that fully implements action semantics as required by VIDE.
PSM Tools available: EMF, ATL Model Weaver, OptimalJ
Selection: EMF, ATL

8.3.4 VIDE Development Tool Requirements: Interoperability of VIDE Technology

REQ – Tool 12	VIDE extensibility	**SHOULD**
The VIDE tools should be extensible via a plug-in mechanism.		

Tools available: Eclipse or EclipseUML, Together, OptimalJ, Objecteering, Rational Enterprise
Selection: Eclipse framework

This requirement implies in particular that a plugin for ensuring quality on model level can be added:

- The visualization of the model (i.e., PIM) should be extensible in order to augment it with information about quality defects (e.g., warn signs).
- The information describing the model (i.e., the PIM) should be accessible from an extension or plugin in order to analyse the model and diagnose quality defects.

REQ – Tool 13	Integration and metadata interchange	**SHOULD**
VIDE should provide model and meta-data interchange capability by adopting the XMI standard		

Tools available: VIDE to provide its capability adhering to MOF's XMI standard; (e.g. Eclipse's EMF, NetBeans MDR) for model and meta-data storage.
Selection: Eclipse framework

8.4 Conclusion

All the observations from the MDA tool review (Section 8.1) and more specifically, the tool-based resource review for supporting the modelling requirements (Section 8.3) serve to show that there is substantial (but not complete or integrated) tool support for many of the development requirements of VIDE. From this review, it is clear that the Eclipse environment offers the greatest degree of support for the development of the VIDE tool. Where VIDE addresses entirely new areas of innovation and model integration, further development or the inclusion of bespoke software components should be made compatible with the Eclipse framework.

9 Implications for the VIDE Architecture

The requirements gathered so far imply requirements on the VIDE architecture, both for the VIDE tools and the language.

9.1 The VIDE Architecture as Contribution to MDSD

VIDE is contributing to Model Driven Software Development as envisaged by OMG's Model Driven Architecture (MDA). It thus follows a strict meta-modelling approach. This entails the separation of:

1. abstract representation as an instance of M2 models and
2. concrete representations as mappings from M2 instances to graphical or textual representations.

We refer to the former as abstract syntax and to the latter as concrete syntax. For a given abstract syntax we may have several concrete syntaxes by defining alternative mappings.
The framework selected for VIDE which supports the definition of abstract syntaxes and mappings to concrete syntaxes is the Eclipse Modelling Framework EMF. With this choice, the VIDE tool:

- has no need for any manual synchronisation between graphical and textual representation of action language sentences—this comes for free with the meta-modelling framework,
- can offer, if needed, a number of concrete syntaxes, both graphical and textual,
- can much more easily provide an editor with syntax completion for the textual concrete syntax.

The following Figure illustrates the meta-modelling architecture.

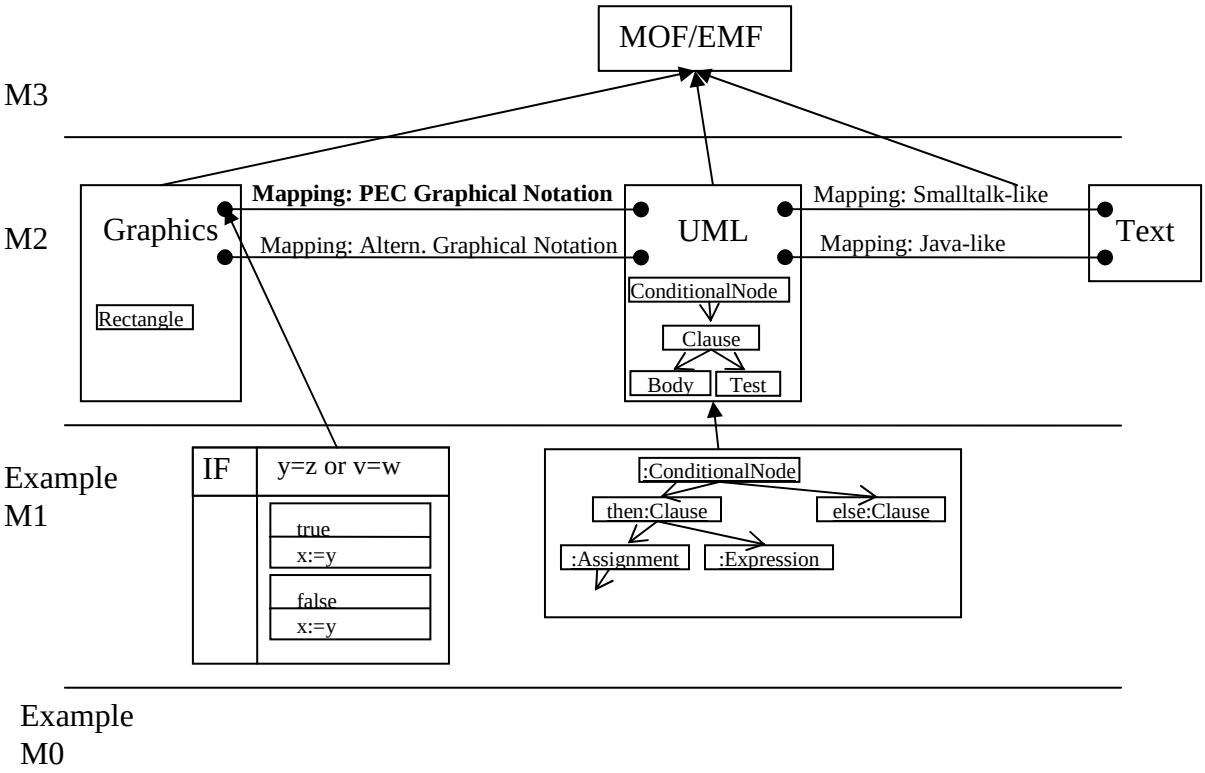


Figure 10 Meta Modelling Architecture

Note that this figure mainly provides a conceptual view on the meta-modelling architecture. To a very large extent, it can be realised technically exactly this way. For transformations between model and text we may, however, utilise specialised techniques as pointed out in the Standards and Languages chapter.
To clarify once again: Any concrete syntax, textual or visual of the VIDE language is just a *view* on the abstract syntax given by UML. As in the classical Model-View-Controller pattern, only the representation in terms of abstract syntax is held in the repository (plus additional information on coordinates of the graphical elements) – changes that occur here trigger changes in all views.

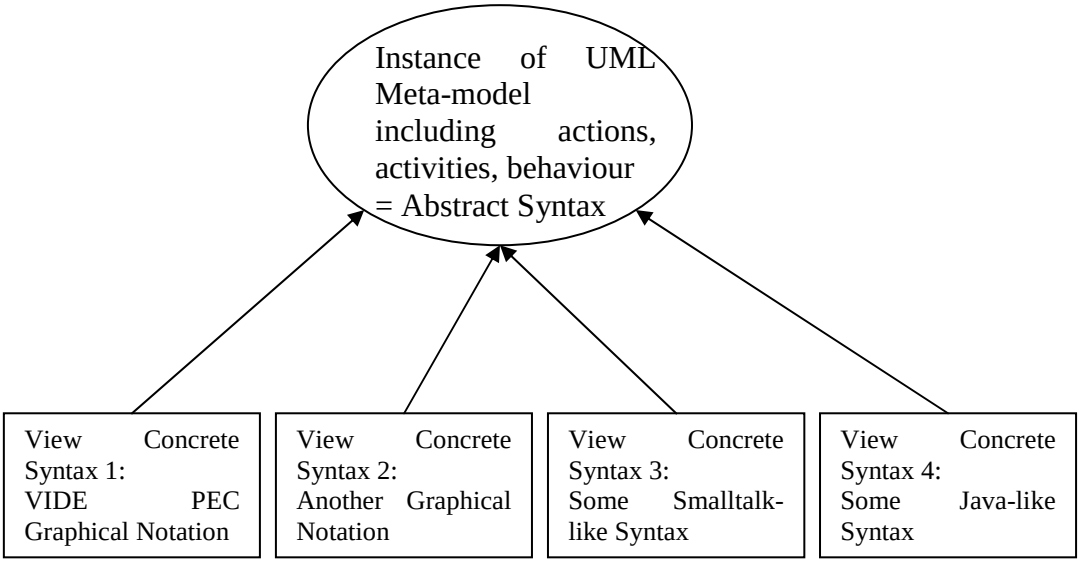


Figure 11 Views on UML Abstract Syntax

REQ – Tool 14	Model driven approach	**MUST**
The VIDE tool strictly follows a model driven approach as stipulated in Figure 10.		
Description: As described in this section.		

Technically, the strict model-driven architecture allows for an easy construction of tools as the following example demonstrates. We have prototyped a VIDE tool with the Eclipse Modelling Framework simply by drawing a meta-model as in Figure 12. It is a simplified excerpt from the UML meta-model. The equivalent tree structure can be seen in

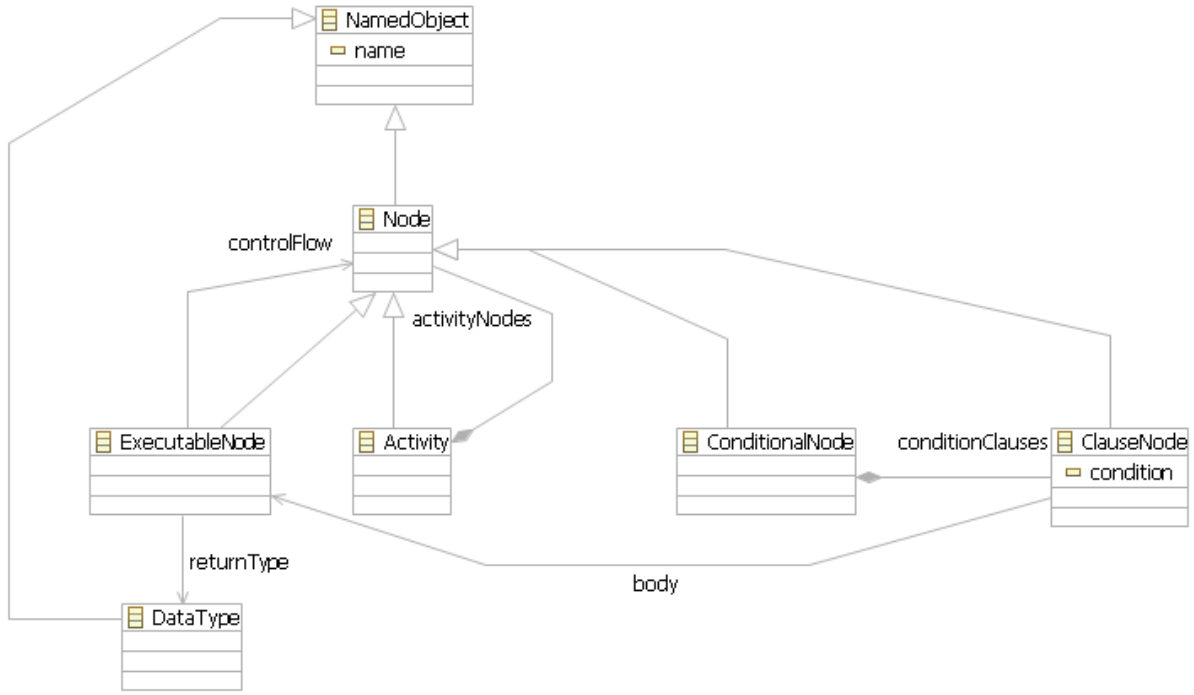


Figure 12: Meta-model of a VIDE Demonstrator

Together with a declarative mapping to the graphical domain, we obtain automatically an editor for an activity diagram like notation of an action language, integrated in Eclipse as can be seen in Figure 13.

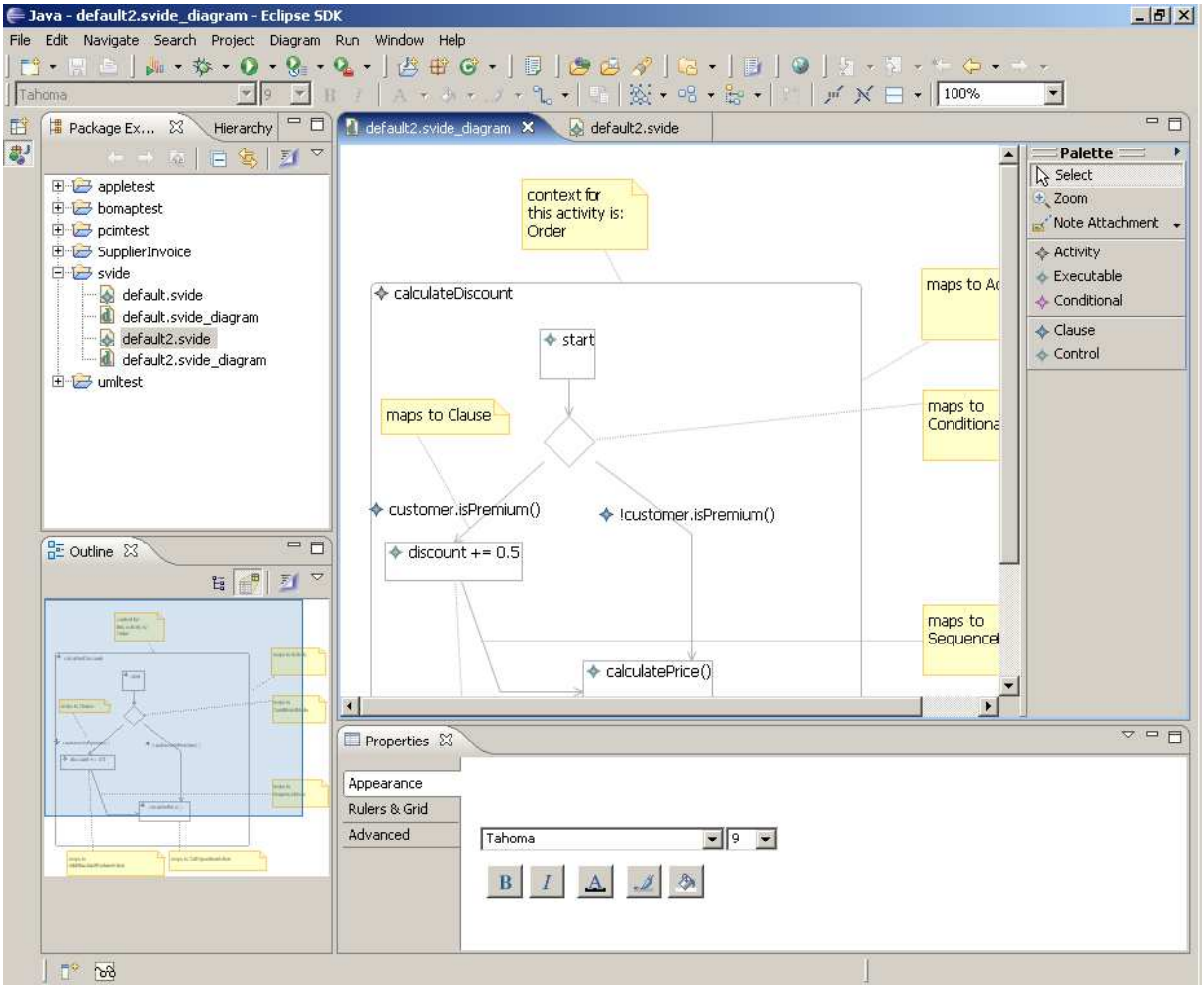


Figure 13: The generated editor

An example model draws with this tool is depicted in Figure 14.

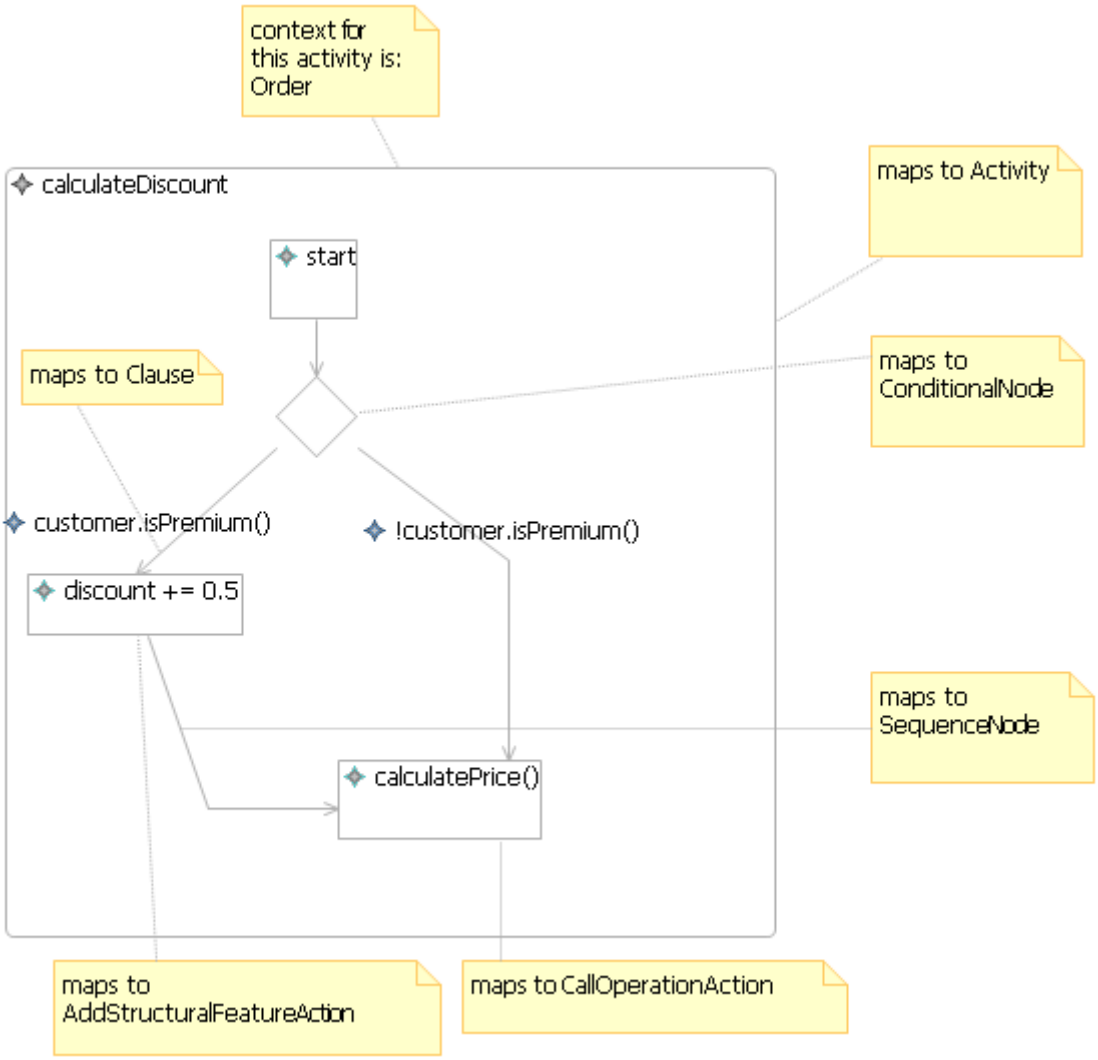


Figure 14: A model drawn with the prototype

9.2 The VIDE Architecture from a User's Point of View

The embedding into the MDSD is reflected in the following overall architecture of the VIDE tool:

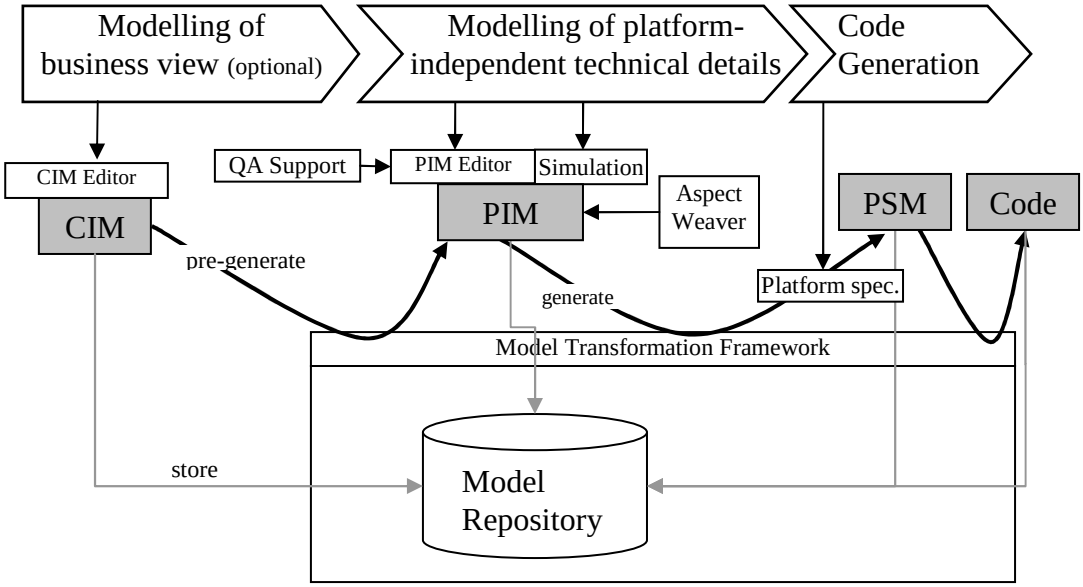


Figure 15: The VIDE architecture from the user's point of view

VIDE users may interact with the system depending on their background and skills either on CIM or on PIM layer. The PSM layer is not touched (or it is read-only). The steps from CIM to PIM layer and from PIM to PSM layer and to code are performed by employing the model transformation framework only. Allowing for modification on the PIM layer, rather than translating from CIM to Code in one step seems a reasonable alleviation to the complexity of the problem as a whole.

REQ – AR1	Architecture	**SHOULD**
Concerning the behaviour from a VIDE user's perspective, the VIDE tool should follow the architecture as depicted in Figure 15.		
Description: See above.		

10 List of Requirements

Requirement Number	Name	Priority
REQ – NonFunc 1	Accessibility at the CIM level	SHOULD
REQ – NonFunc 2	CIM level collaboration	MAY
REQ – NonFunc 3	On-line support for CIM/PIM users	SHOULD
REQ – NonFunc/Semantics 4	Clear and unambiguous notation	SHOULD
REQ – NonFunc 5	Model view saliency	SHOULD
REQ – NonFunc 6	Appropriate textual/graphical fidelity	SHOULD
REQ – NonFunc 7	Timely feedback and constraints	SHOULD
REQ – NonFunc 8	Runnable and testable VIDE prototypes	SHOULD
REQ – NonFunc 9	Scalability of proposed solution.	MUST
REQ – User 1	Flexibility and interoperability of VIDE language and tools	SHOULD
REQ – User 2	Reuse of UML Standard	SHOULD
REQ – Semantics 1	Semantics of VIDE Internal Communication	SHOULD
REQ – Semantics 2	Simple VIDE semantics	SHOULD
REQ – Lang 1	Usage of UML2 Behaviour (“Action Semantics”)	SHOULD
REQ – Lang 2	Simplified UML meta-model	MAY
REQ – Lang 3	User Language & Concepts	SHOULD
REQ – Lang 4	Compliance with Standards	SHOULD
REQ – Lang 5	Deviation from Standards	MAY
REQ – Lang 6	Modularisation and extensibility	SHOULD
REQ – Lang 7	Language for CIM, PIM, PSM modelling	SHOULD
REQ – Tool 1	Usage of Industrially Adopted Tools	MUST
REQ – Tool 2	Meta-modelling Framework	MUST
REQ – Tool 3	Meta-modelling Concepts	SHOULD
REQ – Tool 4	M2M Transformation Technology	SHOULD
REQ – Tool 5	M2T Transformation Technology	SHOULD
REQ – Tool 6	T2M	SHOULD
REQ – Tool 7	Meta-modelling Framework	SHOULD
REQ – Tool 8	Use of OCL	SHOULD
REQ – Tool 9	CIM modelling standards	MAY
REQ – Tool 10	PIM, PSM modelling standards	SHOULD
REQ – Tool 11	Framework for CIM, PIM, PSM modelling	SHOULD
REQ – Tool 12	VIDE extensibility	SHOULD
REQ – Tool 13	Integration and metadata interchange	SHOULD
REQ – Tool 14	Model driven approach	MUST
REQ – AR1	Architecture	SHOULD

11 Glossary

Analyst / Designer: Analysts/Designers are responsible for the conceptual model of business entities and the high level business logic. They use design artefacts and models produced by the business analyst and transforms them into a design. Analysts/Designers work on PIM level in the VIDE tool stack.

Analyst/VIDE Programmer: The Analyst/VIDE Programmer is responsible for the completion of the behavioural model to allow model simulation (i.e. for testing) and the transformation of the models into code. Analysts/VIDE Programmers work on PIM level in the VIDE tool stack.

AOP: Aspect-**O**riented **P**rogramming is a programming paradigm that attempts to aid programmers in the separation of concerns, specifically cross-cutting concerns, to advance the modularization of software. AOP uses crosscutting expressions that encapsulate the concern in one place.

Architect: The architect is responsible for building the transformations of the behavioural models described using VIDE into platform specific coding. The architect is an expert in the target platform (i.e. Struts, …) and the programming language (i.e. Java) but also has a sufficient understanding of UML and VIDE to be able to define the transformation. Architects work on PIM&PSM level in the VIDE tool stack.

ATL: The **A**TLAS **T**ransformation **L**anguage is a result of the MODELWARE project. This transformation language is closely related to the QVT standard and provides a running implementation.

BPMN: Business Process Modelling Notation. The OMG standard BPMN provides a notation that is understandable by business users, including business analysts (creating the initial drafts of the processes), the technical developers (responsible for implementing the technology that will perform those processes), and the business people (who will manage and monitor those processes).

Business Analyst: The Business Analysts advise enterprises on analysis, conception and implementation of IT solutions. They constitute the connection between the customer and the involved IT specialists and need technical as well as social competences. Business Analysts work on CIM level in the VIDE tool stack.

CIM: A **C**omputation **I**ndependent **M**odel represents the user requirements in an abstract, high level view on a software or business system. The transition of a CIM Model into a Platform Independent Model (PIM) should be done automatically using a model transformation.

Domain User (Customer): The Domain User is the end user of the constructed software solution. He works for the customer and is an expert in his special domain typically without knowledge technical issues. The Domain User works on CIM level in the VIDE tool stack.

EMF: Eclipse **M**odeling **F**ramework is a modelling framework for building tools and other applications based on a structured data model. EMF provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. EMF provides the foundation for interoperability with other EMF-based tools and applications.

GEF: Graphical **E**diting **F**ramework allows developers to create a rich graphical editor from an existing application model. Developer can take advantage of many common operations provided in GEF and/or extend them for the specific domain. GEF employs an MVC (model-view-controller) architecture which enables simple changes to be applied to the model from the view.

GMF: The **G**raphical **M**odeling **F**ramework provides a generative component and runtime infrastructure for developing graphical editors based on the Eclipse Modeling Framework (EMF) and Graphical Editing Framework (GEF).

IDE: Integrated **D**evelopment **E**nvironment assists computer programmers in developing software usually consisting of a source code editor, a compiler and/or interpreter, build-automation tools, and a debugger. The VIDE project will extend an existing IDE with tools for describing UML2 Action Semantics

M3/M2/M1 Layers: Metamodelling is defined into a four-layered architecture. The M3 layer provides a meta-meta-model at the top layer. This M3-model is the language used by MOF to build meta-models, called M2-models. These M2-models describe elements of the M1-layer, and thus M1-models. The M0-layer is used to describe the real-world.

MDA: Model-**D**riven **A**rchitecture is a software design approach intended to support model-driven engineering of software systems. MDA was initiated by the OMG.

MDST: Model **D**riven **S**oftware **T**esting derives test cases in whole or in part from a model that describes some (usually functional) aspects of the test system. In VIDE testing should be supported on model (e.g. model simulation) and code level verify the correctness of code transformations.

ModelBus: ModelBus are tools dedicated to model driven development developed by the MODELWARE project. The key feature of ModelBus is possibility to exchange models in heterogeneous formats and a transparent integration of model based tool.

MOF: Meta-**O**bject **F**acility is standard for Model Driven Engineering, proposed by the OMG. MOF provides a meta-meta-model at the top layer and means to create and manipulate models and meta-models. There are two relevant versions of this standard, MOF 1.4 (Object Management Group 2002) and MOF 2.0 (Object Management Group 2004).

OCL: Object **C**onstraint **L**anguage. OCL statements serve as the most precise means of model specification within the UML and MOF model and meta-model definitions. For that purpose OCL was defined to be able to express constraints for any kind of UML elements. OCL moreover provides means to express any (first-order) query on some instance of a UML class diagram.

OMG: Object **M**anagement **G**roup (OMG) is a consortium, originally aimed at setting standards for distributed object-oriented systems, and is now focused on modelling (programs, systems and business processes) and model-based standards in some 20 vertical markets.

Petri Net: Petri Nets are a formal, graphical, executable technique for the specification and analysis of concurrent, discrete-event dynamic systems; a technique undergoing standardization, initially developed by C. A. Petri for the specification of concurrent (parallel) systems.

PIM: A **P**latform **I**ndependent **M**odel is a model of a software or business system that is independent of the specific technological platform used (PSM Level) to implement it. The transition of a PIM Model into a Platform-specific (PSM) model should be done automatically using a model transformation.

PSM: A **P**latform **S**pecific **M**odel is a model of a software or business system that is linked to a specific technological platform (e.g. a specific programming language, operating system or database). The PSM Model should allow for an automatic transformation into code.

Query: A query is the extraction of data from a structured source of information. In the VIDE context, queries are sub-expressions of the VIDE language which extract data from a UML class diagram.

QVT: Query / **V**iews / **T**ransformations is an emerging OMG standard provides technology neutral solutions for querying, transforming and specifying views of MOF-based models.

SDL: The **S**pecification and **D**escription **L**anguage is a specification language for describing system behaviour. Its major use case is in the telecommunication industry for descriptions of process control and real-time applications.

SME: Small & **M**edium-sized **E**nterprises is an abbreviation to classify companies whose headcount or turnover falls below certain limits.

Tefkat: Open source model transformation language developed at Queensland University.

User: A person who interacts with a system.

User Interface (UI): All aspects of a system with which a user can interact and perceive.

UML: Unified **M**odeling **L**anguage is a specification language for object modelling defined at the OMG. UML2 Action Semantics is an essential part of UML 2.0 for the VIDE project.

UML Action semantics: UML Action Semantics refers to the capabilities of UML to describe behaviour algorithmically. UML Action Semantics were in UML 1.4 separated from the rest of UML; since UML 2, one should rather speak of the behavioural part of UML (which is sub-divided in UML actions, activities, and behaviour). Contrary to its name, UML Action Semantics, does primarily define an abstract syntax rather than semantics.

Visual Design: The portion of a user interface that is concerned with the aesthetic quality of an application. Composed of variables that address a specific purpose or function, such as

font, color, and images, which impact the appearance, organization and layout of the graphical elements in a user interface.

XMI: XML **M**etadata **I**nterchange is a MOF-based specification providing the rules of XML serialization of models, allowing their transfer between standard-compliant tools.

12 References

(2006). GMF Tutorial.

Abraham, R. (2003). FoXQ - XQuery by Forms. 2003 IEEE Symposia on Human Centric Computing Languages and Environments, Auckland, New Zealand.

Ahrendt, W., T. Baar, et al. (2005). "The KeY Tool." Software and Systems Modeling **4**(1): 32-54.

Alcatel, I. Logix, et al. (2001). Action Semantics for the UML OMG ad/2001-08-04 Response to OMG RFP ad/98-11-01, Alcatel, I-Logix, Kennedy-Carter, Kabira Technologies Inc., Project Technology Inc., Rational Software Corporation, Telelogic AB.

Allen, E. (2002). Bug patterns in Java. Berkeley, Apress, USA, New York, NY.

Aniszczyk, C. (2005). Using GEF with EMF

Aonix. (2006). "Ameos toolset for MDA (http://www.aonix.com/ameos.html)." Retrieved 08/2006, 2006, from http://www.aonix.com/ameos.html.

Ark, W., D. C. Dryer, et al. (1998). Representation Matters: the Effect of 3D Objects and a Spatial Metaphor in a Graphical User Interface. Proceedings of HCI 98, the Conference on Human-Computer Interaction, Springer.

Atkinson, C. and T. Kühne (2003). Model-Driven Development: A Metamodeling Foundation. IEEE Software. **20(5):** 36-41.

Atkinson, M. and P. Buneman (1987). Types and Persistence in Database Programming Languages. ACM Computing Surveys. **19:** 105-190.

Atkinson, M. and R. Morrison (1995). Orthogonally Persistent Object Systems. The VLDB Journal. **4:** 319-401.

ATLAS group and LINA & INRIA Nantes (2006). ATL: Atlas Transformation Language, ATL User Manual.

Aurum, A., H. Petersson, et al. (2002). "State-of-the-art: software inspections after 25 years." Software Testing, Verification and Reliability, UK * vol 12 (Sept. 2002), no 3, p 133 54, 56 refs.

Avison, D., A.T.Wood-Harper, et al. (1998). "A further exploration into information systems development: the evolution of Multiview2.". IT and People: 124 -138.

Avison, D. and G. Fitzgerald (2006). Information Systems Development: Methodologies, Techniques and Tools. London, UK, McGraw-Hill.

Avison, D. E. and A.T.Wood-Harper (1990). Multiview - an exploration in information systems development. Maidenhead,UK, McGraw-Hill.

B. G. Ryder, M. L. S. M. B. (2005). The Impact of Software Engineering Research on Modern Programming Languages. ACM Transactions on Software Engineering and Methodology. **14:** 431-477.

B. Shneiderman, C. P. (2003). Designing the user interface: Strategies for effective human-computer interaction. Reading, MA, Addison-Wesley.

Baar, T. (2003). "Metamodels without Metacircularities." L'Objet **9**(4): 95--114.

Barbosa, P. A. A., C. F. G. Contreras, et al. (2005). MDA and Separation of Aspects: An approach based on multiple views and Subject Oriented Design. Proc. of 5rd International Workshop on Aspect-Oriented Modeling with UML, AOSD 2005, Chicago, IL.

Barclay, P. J., T. Griffiths, et al. (2003). Teallach - A Flexible User-Interface Development Environment for Object Database Applications. J. Visual Languages and Computing. **14(1):** 47-77.

Barnett, M., K. R. M. Leino, et al. (2005). The Spec# Programming System: An Overview. Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. Berlin / Heidelberg, Springer.

Bennett, K. H. and V. T. Rajlich (2000). Software Maintenance and Evolution: A Roadmap. Future of Software Engineering Track of 22nd ICSE, Limerick, Ireland, ACM Press, New York.

Berry, G. and G. Gonthier (1992). The ESTEREL synchronous programming language: design, semantics, implementation. Science of Computer Programming. **v.19 n.2:** 87-152.

Bézivin, J., S. Hammoudi, et al. (2004). Applying MDA Approach for Web Service Platform. EDOC 2004**:** 58-70.

Blackwell, A. F., M. Burnett, et al. (2004). Champagne Prototyping: A Research Technique for Early Evaluation of Complex End-User Programming Systems. VL/HCC'04: IEEE Symposium on Visual Languages and Human-Centric Computing, Rome, Italy, September 26-29, 2004.

Blanc, X. (2005). ModelBus, MODELWARE.

Blau, H., N. Immelman, et al. (2002). A Visual Language for Querying and Updating Graphs, University of Massachusetts.

Bock, C. (2004). "UML 2 Activity and Action Models Part 4: Object Nodes." Journal of Object Technology **3**(1): 27-41.

Boocock, P. (2006). "The Jamda project (http://jamda.sourceforge.net/)." Retrieved 08/2006, 2006, from http://jamda.sourceforge.net/.

Börger, E. and R. Stärk (2003). Abstract State Machines: A Method for High-Level System Design and Analysis. Heidelberg, Springer.

Borland. (2006). "Together Architect (http://www.borland.com/us/products/together/index.html)." Retrieved 07/2006, 2006, from http://www.borland.com/us/products/together/index.html.

Braubach, L., A. Pokahr, et al. (2002). Tool-Supported Interpreter-Based User Interface Architecture for Ubiquitous Computing. Interactive Systems - Design, Specification, and Verification. Q. L. B. U. J. V. P. Forbrig, Springer Heidelberg**:** 89--103.

Bray, I. (2002). An Introduction to Requirements Engineering. Harlow, UK, Pearson Education Limited.

Brown, W. J. (1998). AntiPatterns: refactoring software, architectures, and projects in crisis, Wiley.

Bruntink, M., D. A. van, et al. (2004). "An evaluation of clone detection techniques for crosscutting concerns." Proceedings. 20th IEEE International Conference on Software Maintenance, Chicago, IL, USA, 11 14 Sept. 2004 * Los Alamitos, CA, USA: IEEE Comput. Soc, 2004, p 200 9.

Brykczynski, B. (1999). "A survey of software inspection checklists." Software Engineering Notes **24**(1): 82-89.

Buck-Emden, R. and P. Zencke (2004). mySAP CRM: The Official Guidebook to SAP CRM 4.0, Galileo Press.

Budinsky, F., D. Steinberg, et al. (2004). Eclipse Modeling Framework, Addison Wesley Professional.

C.J. Date, H. D. (1992). Relational Database Writings 1989-1991, Addison-Wesley.

Campbell, C., W. Grieskamp, et al. (2005). Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. Redmond, Microsoft Research.

Cardelli, L. and P. Wegner (1985). On Understanding Types, Data Abstraction and Polymorphism. ACM Computing Surveys. **17:** 471-522.

Carlisle, M., T. Wilson, et al. (2005). RAPTOR: a Visual Programming Environment for Teaching Algorithmic Problem Solving. Proceedings of the 36th SIGCSE technical symposium on Computer science education**:** 176 â€“ 180.

Catarci, T. (2000). What Happened When Database Researchers Met Usability. Information Systems. 25(3): 177-212.

Cattel, R. G. G. (1994). Object Data Management, Addison-Wesley.

Cattel, R. G. G. and D. K. B. Ed (2000). Object Data Management Group: The Object Database Standard ODMG, Release 3.0, Morgan Kaufmann.

Chavez, C. and C. Lucena A Metamodel for Aspect-Oriented Modeling.

Checkland, P. B. (1999). Soft Systems Methodology in Action, John Wiley and Sons Ltd.

Chitchyan, R., A. Rashid, et al. (2005). Report synthesizing state-of-the-art in aspect-oriented requirements engineering, architectures and design, Lancaster University, AOSD-Europe Deliverable D11, AOSD-Europe-ULANC-9.

Ciolkowski, M., O. Laitenberger, et al. (2002). Software inspections, reviews and walkthroughs. Proceedings of the 24th International Conference on Software Engineering. ICSE 2002, New York, NY, USA.

Clarke, S. a. and R. Walker Towards a Standard Design Language for AOSD.

Codagen. (2006). "Codagen Architect for MDA (http://www.codagen.com/)." Retrieved 07/2006, 2006, from http://www.codagen.com/.

Compuware. (2006). "OptimalJ MDA tool (http://www.compuware.com/products/optimalj/)." Retrieved 08/2006, 2006, from http://www.compuware.com/products/optimalj/.

Crowle, S. (2004). Into the mangle: Software engineers run creases through a user interface metaphor. Developing User Interfaces with XML: Advances on User Interface Description Languages workshop, Advanced Visual Interfaces, AVI'2004.

Czerwinski, M. (2002). Handbook of HCI. J. Jacko and A. Sears, Erlbaum: NJ**:** 19-21.

Date, C. J. (1986). Relational Database: Selected Writings, Addison-Wesley.

Demeyer, S., S. Ducasse, et al. (2003). Object-oriented reengineering patterns. San Francisco, Morgan Kaufman Publishers.

DotNetBuilders. (2006). "Constructor toolset for MDA (http://www.dotnetbuilders.com/constructor.aspx)." Retrieved 07/2006, 2006, from http://www.dotnetbuilders.com/constructor.aspx.

Easterbrook, S. M. (1991). Elicitation of Requirements from Multiple Perspectives. London, University of London.

Eclipse. (2006). "Eclipse project (http://www.eclipse.org/)." Retrieved 08/2006, 2006.

Eichberg, M. (2002). MDA and Programming Languages. Proceedings of the Workshop on Generative Techniques in the context of Model Driven Architecture. OOPSLA, November 2002.

Erwin, M. (2003). Xing: A Visual XML Query Language. Journal of Visual Languages and Computing. **14(1):** 5â€“45.

Faulkner, X. (2000). Usability Engineering. London, Macmillan Press Ltd.

Fenton, N. E. and M. Neil (1999). "Software metrics: successes, failures and new directions." Journal of Systems and Software **47**(2-3): 149-57.

Fenton, N. E. and N. Ohlsson (2000). Quantitative analysis of faults and failures in a complex software system. IEEE Transactions on Software Engineering. **26, no. 8:** 797-814.

Filman, R. E. What Is Aspect-Oriented Programming, Revisited.

Fowler, M. (1999). Refactoring: Improving the Design of Existing Code, Addison-Wesley.

Freimut, B. (2001). Developing and Using Defect Classification Schemes. Fraunhofer IESE Report. Kaiserslautern, Fraunhofer IESE**:** 44.

Fuentes, L. and P. Sanchez Elaborating UML 2.0 Profiles for AO Design.

G.Kiczales, J. Lamping, et al. (1997). Aspect-Oriented Programming. Proc. ECOOP Conf.1997, Springer. **1241:** 220-242.

Gardner, T., C. Griffin, et al. (2003). A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard, OMG.

Gartner, I. (2006). "Hype Cycle for Emerging Technologies, 2006."

Gasevic, D., D. Djuric, et al. (2005). Bridging MDA and OWL Ontologies. J. Web Eng. **4(2):** 118-143.

Gause, D. C. and G. M. Weinberg (1989). Exploring Requirements: Quality Before Design. New York, US.

Génova, G., M. C. Valiente, et al. (2005). A Semiotic Approach to UML Models. Advanced Information Systems Engineering, 17th Int. Conf, CAiSE'05, Portugal, FEUP.

Glass, R. (1998). Software Runaways. Harlow, Prentice Hall.

Go, K. and J. M. Carroll (2004). The Blind Men and the Elephant: Views of Scenario-Based System Design. interactions. **11:** 45-53.

Gordon, D., R. Biddle, et al. (2003). A Technology for Lightweight Web-Based Visual Applications, Victoria University of Wellington - School of Mathematical and Computing Science.

Gosling, J., B. Joy, et al. (2000). The Java Language Specification, Addison Wesley.

Green, T. R. G. and M. Petre (1996). "Usability analysis of visual programming environments: A 'cognitive dimensions' framework." Journal of Visual Languages and Computing **7**: 131-174.

Griffiths, T., P. J. Barclay, et al. (2001). Teallach: a model-based user interface development environment for object databases. Interacting with Computers. **14:** 31-68.

Groher, I. and S. Schulze Generating Aspect Code from UML Models.

Guibert, N., P. Girard, et al. (2004). Example-based programming: a pertinent visual approach for learning to program. Proceedings of the working conference on Advanced visual interfaces, Gallipoli, Italy 2004**:** 358 â€“ 361.

Gunter, C. (1992). Semantics of Programming Languages, MIT Press.

Gurevich, Y., B. Rossman, et al. (2005). Semantic essence of AsmL, Elsevier Science Publishers Ltd. **343:** 370-412.

Hall, T., S. Beecham, et al. (2002). Requirements problems in twelve software companies: an empirical analysis. 6th International Conference on Empirical Assessment in Software Engineering, Keele, Keele University.

Han, Y., G. Kniesel, et al. (2005). Towards Visual AspectJ by a Meta Model and Modeling Notation. Proc. of 5rd International Workshop on Aspect-Oriented Modeling with UML, AOSD 2005, Chicago, IL.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. Science of Computer Programming. **8:** 231-274.

Haustein, S. and J. Pleumann (2004). OCL as Expression Language in an Action Semantics Surface Language. Workshop on OCL and Model Driven Engineering at Seventh International Conference on UML Modeling Languages and Applications. Lisbon.

Hendry, D. G. (2004). Communication Functions and the Adaption of Design Representations in Interdisciplinary Teams. Designing Interactive Systems, Cambridge, Massachusetts, ACM.

Hendry, D. G. (2006). Sketching with Conceptual Metaphors to Explain Computation Processes. Proceedings of IEEE Symposium on Visual Languages/Human-Centric Computing 2006, Brighton, UK, IEEE Computer Society Press.

Hundausen, C. D. (2001). Communicative Dimensions of End-User Environments. Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01) IEEE Computer Society.

IBM. (2006). "Rational XDE Developer (http://www-306.ibm.com/software/uk/rational/awdtools/swdeveloper.html)." Retrieved 08/2006, 2006, from http://www-306.ibm.com/software/uk/rational/awdtools/swdeveloper.html.

IBM. (2006). "Websphere MDA tool (http://www-306.ibm.com/software/websphere/)." Retrieved 08/2006, 2006, from http://www-306.ibm.com/software/websphere/.

International Communication Union (2002). Specification and Description Language (SDL). Recommendation Z.100 (08/2002), International Communication Union.

Jackson, A. and S. n. Clarke (2006). Initial Version of Aspect-Oriented Design Approach, Trinity College Dublin, AOSD-Europe Deliverable D38, AOSD-Europe-TCD-7.

Jackson, M. (1995). Software Requirements & Specifications:a lexicon of practice, principles and prejudices, Addison-Wesley.

Jensen, K. (1997). Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Heidelberg, Springer.

K.Subieta (2004). Theory and construction of object-oriented query languages, PJIIT - Publishing House.

K.Subieta (2006). Stack-Based Approach (SBA) and Stack-Based Query Language (SBQL).

K.Subieta, Y. Kambayashi, et al. (1995). Procedures in Object-Oriented Query Languages. Proc. 21-st VLDB Conf., Zurich, 1995**:** 182-193.

K.T.Phalp and K. Cox (2003). Using Enactable Models to Enhance Use Case Descriptions. ProSim'03, International Workshop on Software Process Simulation Modelling (in conjunction with ICSE 2003), Portland, USA, May 3-4 2003.

Kang, H., C. Plaisant, et al. (2003). New Approaches to Help Users Get Started with Visual Interfaces: Multi-Layered Interfaces and Integrated Initial Guidance. Proc. 2003 National Conference on Digital Government Research**:** 141-146.

Kanyaru, J. and K. Phalp (2005). Requirements validation with enactable models of state-based use cases. Empirical Assessment in Software Engineering, EASE 2005, Keele University, 11-13 April 2005.

Keller, G., M. Nüttgens, et al. (1992). Semantische Prozessmodellierung auf der Grundlage "ereignisgesteuerter Prozessketten (EPK). Veröffentlichungen des Instituts für Wirtschaftsinformatik, Universität des Saarlandes**:** 2.

Keller, S. (2000). Entwicklung einer Methode zur integrierten Modellierung von Strukturen und Prozessen in Produktionsunternehmen. Düsseldorf, VDI Verlag.

Kiesner, C., G. Taenzer, et al. (2002). Visual OCL: A visual notation for the Object Coonstraint Language. Forschungsberichte der Fakultaet 4 Elektotechnik und Informatik der TU Berlin.

Kim, S.-K. and D. Carrington (1999). Formalizing the UML Class Diagram Using Object-Z. "UML" '99 - The Unified Modeling Language: Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 1999. Proceedings. R. France and B. Rumpe. Berlin / Heidelberg, Springer.

Kitchenham, B. A. and L. Jones (1997). "Evaluating Software Engineering Methods and Tool, Part 6: Identifying and Scoring Features." Software Engineering Notes **22**(2): 16-18.

Klaas van den Berg, J. M. C. R. C. (2005). AOSD Ontology 1.0 - Public Ontology of Aspect-Orientation, AOSD-Europe-UT-01, D9, AOSD-Europe.

Kleppe, A. G., J. Warmer, et al. (2003). MDA Explained: The Model Driven Architecture: Practice and Promise, Addison-Wesley Longman Publishing Co., Inc.

Ko, A. J., B. A. Myers, et al. (2004). Six learning barriers in end-user programming systems. Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing 2004, Rome, Italy, IEEE Computer Society Press.

Kovse, J. and T. Härder (2004). MT-Flow - An Environment for Workflow-Supported Model Transformations in MDA. CAiSE 2004**:** 160-174.

Kozankiewicz, H., J. Leszczylowski, et al. (2003). New Approach to View Updates. Proc. of the Emerging Database Research in Eastern Europe VLDB Workshop, Berlin, Germany, 2003.

Kules, W. and B. Shneiderman (2003). Designing a Metadata-Driven Visual Information Browser for Federal Statistics. Proceedings of the 2003 National Conference on Digital Government Research**:** 117-122.

Kulkarni, V. and S. Reddy (2003). Supporting Aspects in MDA. Proc. of the Workshop in Software Model Engineering on the UML'2003, San Francisco, USA, 2003.

Lawley, M. J. and J. Steel (2005). Practical Declarative Model Transformation With Tefkat. Satellite Events at the MoDELS 2005 Conference. Jamaica, Springer.

Lawrance, J., S. Clarke, et al. (2005). How Well Do Professional Developers Test with Code Coverage Visualizations? VL/HCC'05: IEEE Symposium on Visual Languages and Human-Centric Computing, September 2005.

Leffingwell, D. and D. Widrig (2003). Managing Software Requirements: A Use Case Approach. Boston, US, Addison-Wesley.

Leonhardt, U. (1995). Decentralised process enactment in a multi-perspective development environment. 17th international conference on Software engineering, Seattle, Washington, United States, ACM Press.

Leopold, J. L., M. Heimovics, et al. (2002). Webformulate: a Web-based visual continual query system. Proceedings of the Eleventh International World Wide Web Conference (WWW2002), Honolulu, Hawaii, USA**:** 221-231.

Liggesmeyer, P. (2003). "Testing safety-critical software in theory and practice: a summary." IT Information Technology **45**(1): 39-45.

Luyten, K., T. Clerckx, et al. (2003). Derivation of a Dialog Model from a Task Model by Activity Chain Extraction. Interactive Systems Design Specification, and Verification : 10th International Workshop, DSV-IS 2003**:** 203-217.

M.Lentner, K. Stencel, et al. (2006). Semi-Strong Static Type Checking of Object-Oriented Query Languages. Proc. of 32nd International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM'06, Springer. **3831:** 399-408.

Mantyla, M., J. Vanhanen, et al. (2003). "A taxonomy and an initial empirical study of bad smells in code." International Conference on Software Maintenance, Amsterdam, Netherlands, 22 26 Sept. 2003 * Los Alamitos, CA, USA: IEEE Comput. Soc, 2003, p 381 4.

Mantyla, M. V., J. Vanhanen, et al. (2004). "Bad smells - humans as code critics." Proceedings. 20th IEEE International Conference on Software Maintenance, Chicago, IL, USA, 11 14 Sept. 2004 * Los Alamitos, CA, USA: IEEE Comput. Soc, 2004, p 399 408.

Marcus, A., L. Feng, et al. (2003). 3D Representations for Software Visualization. Proceedings of the ACM Symposium on Software Visualization (SoftVis 2003), San Diego, CA, June 11-13**:** 27-36.

Matula, M. (2003). "NetBeans Metadata Repository." 2006.

Mayers, B. A., A. J. Ko, et al. (2006). Invited Research Overview: End-User Programming. CHI 2006, April 22-27, 2006, Montreal, Canada, ACM 1-59593-294-4/06/0004.

McCrickard, D. S., M. Czerwinski, et al. (2003). Introduction: design and evaluation of notification user interfaces. International Journal of Human-Computer Studies. **58:** 509-514.

Mellor, S. J. and M. J. Balcer (2002). Executable UML: A Foundation for Model-Driven Architecture, Addison Wesley.

Mellor, S. J. and K. Scott, et al. (2004). MDA Distilled: Principles of Model-Driven Architecture, Addison Wesley.

Mellor, S. J. and A. Watson (2006). "Roles in the MDA Process."

Melton, J., A. R. Simon, et al. (2001). SQL:1999 - Understanding Relational Language Components, Morgan Kaufmann Publishers.

Mens, T. and T. Tourwe (2004). "A survey of software refactoring." IEEE Transactions on Software Engineering, USA * vol 30 (Feb. 2004), no 2, p 126 39, 111 refs.

Mezini, M. and K. Ostermann (2005). A Comparison of Program Generation with Aspect-Oriented Programming. Proc. of the EU-NSF Strategic Research Workshop on Unconventional Programming Paradigms, Springer Verlag. **3566**.

Millot, P. (2004). MODELWARE fact sheet.

Müller, P. and A. Poetzsch-Heffter (2000). A Type System for Controlling Representation Exposure in Java. Workshop on Formal Techniques for Java Programs at ECOOP.

NetBeans. (2006). "MDR resources (http://mdr.netbeans.org/)." 2006.

NetBeans. (2006). "NetBeans (http://www.netbeans.org/; http://www.netbeans.org/about/press/8.html)." Retrieved 07/2006, 2006.

Norman, D. A. (2005). Human-Centered Design Considered Harmful. interactions. **12:** 14-19.

Nuseibeh, B., J. Kramer, et al. (2003). ViewPoints: Meaningful Relationships Are Difficult! Proceedings of International Conference on Software Engineering(ICSE'03), Portland, Oregon, USA, IEEE CS Press.

Object Management Group (2001). Model Driven Architecture (MDA).

Object Management Group (2002). Meta Object Facility (MOF) Specification, Version 1.4.

Object Management Group (2002). OMG CORBA/IIOP Specifications.

Object Management Group (2002). UML 1.4 with Action Semantics. Final Adopted Specification.

Object Management Group (2003). UML 2.0 Object Constraint Language Specification.

Object Management Group (2003). Common Warehouse Metamodel (CWM) Specification, Version 1.1.

Object Management Group (2004). Meta Object Facility (MOF) Core Specification, Version 2.0.

Object Management Group (2004). UML 2.0 Infrastructure Specification.

Object Management Group (2004). UML 2.0 Superstructure Specification.

Object Management Group (2005). MOF QVT Final Adopted Specification.

Object Management Group (2005). Semantics of a Foundational Subset for Executable UML Models RFP.

Object Management Group (2005). XML Metadata Interchange (XMI) Specification. Version 2.0.

Object Management Group (2006). BPMN 1.0: OMG Final Adopted Specification.

Objecteering. (2006). "Objecteering/UML (http://www.objecteering.com/)." Retrieved 07/2006, 2006, from http://www.objecteering.com/.

Objects, I. (2006). "ArcStyler MDA tool (http://www.arcstyler.com/)." Retrieved 08/2006, 2006, from http://www.arcstyler.com/.

Oldevik, J. (2006). MOFScript User Guide.

PathFinderSolutions. (2006). "PathMate MDA transformation environment (http://www.pathfindermda.com/products/index.php)." Retrieved 07/2006, 2006, from http://www.pathfindermda.com/products/index.php.

Petri, C. A. (1962). Kommunikation mit Automaten, University Bonn.

Pfleeger, S. (2005). Software engineering: theory and practice, Prentice Hall.

Phalp, K. and K. Cox (2002). Supporting Communicability with Use Case Guidelines: An Empirical Study. 6th International Conference on Empirical Assessment and Evaluation in Software Engineering, Keele University, Staffordshire, UK.

Preece, J., Y. Rogers, et al. (2002). Interaction Design. New York, John Wiley & Sons, Inc.

Rashid, A., A. Garcia, et al. (2006). Aspect-Oriented Software Development Beyond Programming. Proceedings of 9th International Conference on Software Reuse, Torino, Italy.

Reynolds, J. C. (1998). Theories of Programming Languages, Cambridge University Press.

Richters, M. (2002.). A Precise Approach to Validating UML Models and OCL Constraints. Logos Verlag, Berlin, BISS Monographs, No. 14, Universitaet Bremen.

Riel, A. J. (1996). Object-oriented Design Heuristics. Reading, Mass., Addison-Wesley Pub. Co.

Robertson, S. (2001). "Requirements trawling: techniques for discovering requirements." Int. Journal of Human-Computer Studies **55**: 405-421.

Roe, D., K. Broda, et al. (2003). Mapping UML Models incorporating OCL Constraints into Object-Z. London, Imperial College.

Roock, S. and M. Lippert (2004). Refactorings in großen Softwareprojekten: Komplexe Restrukturierungen erfolgreich durchführen. Heidelberg, dpunkt Verlag.

Rosson, M. B., J. Ballin, et al. (2004). Everyday Programming: Challenges and Opportunities for Informal Web Development. IEEE Symposium on Visual Languages and Human-Centric Computing, Rome, Italy, September 2004**:** 123-130.

Schauerhuber, A., W. Schwinger, et al. Towards a Common Reference Architecture for Aspect-Oriented Modeling.

Scheer, A.-W. (1999). ARIS, Business Process Framework. Berlin, .

Schmitt, P. H. (2001). A Model Theoretic Semantics of OCL. IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development. Siena, Italy, Technical Report DII 07/01, Dipartimento di Ingegneria
dell'Informazione, Universit\`a degli Studi di Siena.

Selic, B., G. Gullekson, et al. (1992). ROOM: An Object-Oriented Methodology for Developing Real-Time Systems. Proc. 5th International Workshop on Computer-Aided Software Engineering.

Shlaer, S. and S. J. Mellor (1991). Object Lifecycles: Modeling the World in States, Yourdon Press.

Simon, F., F. Steinbruckner, et al. (2001). Metrics based refactoring. Fifth European Conference on Software Maintenance and Reengineering (CSMR), Los Alamitos, CA, USA, IEEE Comput. Society

Soc. Tech. Council on Software Eng (TCSE)
Reeng. Forum
Inst. Superior Tecnico
University Nova de Lisboa.

Smith, M. and P. King (2002). The Exploratory Construction Of Database Views, School of Computer Science and Information Systems, Birkbeck College, University of London.

Sowa, J. W. and J. A. Zachman (1992). "Extending and Formalizing the Framework for Information Systems Architecture." IBM Systems Journal, IBM Publication G321-5488. 1-800-879-2755 **31**.

Stärk, R., J. Schmid, et al. (2001). Java and the Java Virtual Machine — Definition, Verification, Validation. Heidelberg, Springer.

Stavness, N. and K. Shneider (2004). Supporting Workflow in User Interface Description Languages. Developing User Interfaces with XML: Advances on User Interface Description Languages.

Stein, D., S. Hanenberg, et al. (2004). Modeling Pointcuts. Proc. of the 7th International Conference on the Unified Modeling Language (UML 2004), Lisbon, Portugal, October 11-15, 2004, Springer. **3273:** 98-112.

Stencel, K. (2006). Semi-strong Type Checking in Database Programming Languages, PJIIT - Publishing House.

Stolte, C., D. Tang, et al. (2002). Polaris: A System for Query, Analysis, and Visualization of Multidimensional Relational Databases. IEEE Transactions on Visualization and Computer Graphics, IEEE Computer Society. **08:** 52-65.

Subieta, K. (2004). Theory and construction of object-oriented query languages, PJIIT - Publishing House.

Subieta, K. (2006). Stack-Based Approach (SBA) and Stack-Based Query Language (SBQL).

Subieta, K., C. Beeri, et al. (1995). A Stack-Based Approach to Query Languages. Proc. of 2nd East-West Database Workshop, 1994, Springer Workshops in Computing, 1995.

Subieta, K., Y. Kambayashi, et al. (1995). Procedures in Object-Oriented Query Languages. Proc. 21-st VLDB Conf., Zurich, 1995**:** 182-193.

Sun Microsystems, I. (2004). "NetBeans Metadata Repository (MDR)." from. http://mdr.netbeans.org/.

Sutcliffe, A. and N. Maiden (1993). Use of Domain Knowledge for Requirements Validation. Proceedings of IFIP WG 8.1 Conference on Information System Development Process, Elsevier Science Publishers.

Tahvildari, L., K. Kontogiannis, et al. (2003). "Quality-driven software re-engineering." Journal of Systems and Software **66**(3): 225-39.

The Workflow Management Coalition Specification (2005). Workflow Management Coalition Workflow Standard Process Definition Interface
-- XML Process Definition Language.

Tourwe, T. and T. Mens (2003). "Identifying refactoring opportunities using logic meta programming." IEEE Comput: Reengneering Forum; Univ. Sannio. - In Proceedings Seventh European Conference on Software Maintenance and Reengineering. - Los Alamitos, CA, USA, USA IEEE Comput. Soc, 2003, xi+420 91-100, 31 Refs.

Traetteberg, H. (2003). Dialog Modelling with Interactors and UML Statecharts - A Hybrid Approach. Interactive Systems Design Specification, and Verification : 10th International Workshop, DSV-IS 2003**:** 346-361.

Trzaska, M. and K. Subieta (2004). Usability of Visual Information Retrieval Metaphors for Object-Oriented Databases. Proceedings of the On The Move Federated Conferences and Workshops (DOA, ODBASE, CoopIS, PhD Symposium), Larnaca, Cyprus, 2004, Springer. **3292:** 822-833.

van Emden, E. and L. Moonen (2002). "Java quality assurance by detecting code smells." Reengineering Forum; Virginia Commonwealth Univ.; IEEE Comput: Burd, E.. - Los Alamitos, CA, USA, USA IEEE Comput. Soc, 2002, x+349 97-106, 25 Refs.

Vernadat, F. (1996). "Enterprise Modeling and Integration."

Völter, M. (2005). Models and Aspects. Patterns for Handling Cross-Cutting Concerns in the context of MDSD.

Wampler, D. (2003). The Role of Aspect-Oriented Programming in OMG's Model-Driven Architecture, Aspect Programming, Inc.

Warmer, J. and A. Kleppe (1999). The object constraint language: precise modeling with UML, Addison-Wesley Longman Publishing Co., Inc.

Wegmann, A. and O. Preiss (2003). MDA in Enterprise Architecture? The Living System Theory to the Rescue. EDOC 2003**:** 2-13.

White, S. A. (2004). "Introduction to BPMN."

Whitmire, S. A. (1997). Object-oriented Design Measurement. New York, NY, USA, John Wiley & Sons.

Wilkie, I. and King, A. et al. (2001). UML ASL Reference Guide. Kennedy Carter Limited.

Wohlin, C., A. Aurum, et al. (2002). "Software inspection benchmarking-a qualitative and quantitative comparative opportunity." Proceedings Eighth IEEE Symposium on Software Metrics, Ottawa, Ont., Canada, 4 7 June 2002 * Los Alamitos, CA, USA: IEEE Comput. Soc, 2002, p 118 27.

Xactium. (2006). "XMF Mosaic (http://albini.xactium.com/web/index.php?option=com_content&task=blogcategory&id=27&Itemid=46)." Retrieved 07/2006, 2006, from http://albini.xactium.com/web/index.php?option=com_content&task=blogcategory&id=27&Itemid=46.

Xerox AspectJ Programming Guide, Xerox Corporation.

Zachman, J. A. (1987). "A Framework for Information Systems Architecture." IBM Systems Journal **26**.

Zdonik, S. B. and D. Maier, Eds. (1990). Fundamentals of Object-Oriented Databases. Readings in Object-Oriented Database Systems. San Mateo, CA, Kaufmann.

Zhang, G. (2005). Towards Aspect-Oriented Class Diagrams. Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05), pp. 763-768, Taipei, Taiwan.