SPECIFIC TARGETED RESEARCH PROJECT
INFORMATION SOCIETY TECHNOLOGIES

FP6-IST-2005-033606

VIsualize all moDel drivEn programming
VIDE

WP 6	**Deliverable number D.6.1** **Model Compilers**

Project name:	Visualize all model driven programming
Start date of the project:	01 July 2006
Duration of the project:	30 months
Project coordinator:	Polish-Japanese Institute of Information Technology
Leading partner:	SOFTEAM
Due date of deliverable:	31.01.2008
Actual submission date	15.04.2008
Status	developed / draft / **final**
Document type:	Report
Document acronym:	DEL
Editor(s)	François Jaouen, Anis Charfi, Piotr Habela, Krzysztof Stencel, Marcin Daczkowski
Reviewer(s)	François Jaouen, Piotr Habela, Anis Charfi
Accepting	Kazimierz Subieta
Location	www.vide-ist.eu
Version	1.0
Dissemination level	**PU**/PP/RE/CO

Abstract:

The VIDE project aims at a visual, Unified Modeling Language (UML) compliant action language, the VIDE language, suited to business applications. The language is to be used in the model driven software development process (which raises the requirements of its standard-compliance). Further development of the project also includes the integration of a business oriented modelling, aspect-oriented facilities, and means for quality assurance provided inside a powerful, platform-independent development toolset.

This document specifies the mapping of the VIDE metamodel, which is compliant with UML and OCL metamodel to Java and ODRA prototype ODBMS with its innovative SBQL (Stack Based Query Language) developed by PJIIT.

The mapping integrates advanced features like the declaration and the consumption of WSDL based Web Services and RDBMS queries.

Finally this document will propose some ideas of improvement for the UML metamodel definition.

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

History of changes

Date	*Version*	*Author*	*Change description*
14.11.2007	0.1	F. Jaouen	Document creation
02.04.2008	0.2	F. Jaouen, A. Spriestersbach, A. Charfi	VIDE Metamodel presentation, mapping action to Java, mapping activities to Java, edition.
03.04.2008	0.3	F. Jaouen P. Habela	Mapping VIDE to ODRA chapter incorporated, explanation of VIDE language design choices, edition
04.04.2008	0.4	F. Jaouen, P. Habela, A. Charfi	VIDE metamodel presentation, Study of implementation tools for Java compiler edition
11.04.2008	0.5	F. Jaouen, P. Habela, A. Charfi	ODRA presentation, mapping Activity to Java, edition
14.04.2008	0.7	F. Jaouen, P. Habela, A. Charfi	Mapping to JPA, mapping to web services, edition
15.04.2008	1.0	F. Jaouen, P. Habela, A. Spriestersbach, A. Charfi	Final editing

Executive summary

This document describes the mapping of VIDE language to two execution platforms: Java and ODRA. The first is a well known general purpose object oriented programming language while the other belongs to a new brand of object oriented programming language that integrates database query to its core and designated for rapid development of business intensive application.

This work is based on WP1 that has collected requirements and WP2 that has defined the VIDE metamodel, the starting point of the mappings.

The mapping to Java includes three steps:

1. Mapping to plain Java presented metaclasses by metaclasses and structured around the 4 packages: Structures, Activities, Actions and Expressions.
2. Mapping to JPA to allow VIDE program to interact transparently with databases. This mapping integrates creation and deletion of persistent object as well as navigating through persistent object and define mapping for object queries based on JPQL.
3. Mapping to web services, using the annotations defined in JAX-WS standard API. The mapping to web services is bidirectional: the compiler can generate code to produce web services as well as generating code to call externally defined web services.

Mapping the SBQL language used in the ODRA system in turn, exemplifies a transformation to a more homogeneous target platform. ODRA is a purely object-oriented environment that provides a seamlessly integrated query and programming language. A similar approach is followed by the VIDE language, where the behavioural constructs of UML are integrated with the expression language part represented by OCL that provides a powerful querying capability. Development of that mapping has several purposes. Firstly, it prevents VIDE unintentional becoming a Java-only solution. Secondly, it will allow to check, what mapping problems are inherent to code generation in general, and what of them are rather related with the object-relational interaction complexity. Thirdly, by confronting common OMG modelling specifications with the concepts of this ODBMS prototype, it provides insight into the problem of specifying platform-neutral foundation for an object-oriented database management system standard.

This document also contains a study of available tools to implement the mapping to Java. It concludes that OpenArchitectureWare is the best choice according to the requirements established in WP1 (integration to Eclipse, Xpand template based transformation tool).

One goal of the study is propose some enhancements of the UML metamodel. This is done in the last chapter of this document.

Table of Contents

List of Tables

List of Figures

1 Introduction and Overview

This deliverable builds onto the UML 2.1 and OCL 2.0 metamodels and their underlying semantics (as defined in standard specifications and refined in [VIDE2007c]). It is intended to specify rules of transforming VIDE-created UML models into executable code on chosen exemplary target platforms. One of them – Java – was chosen so as to investigate and develop VIDE support for highly automated development of code for popular industrially used platforms. This unit of work also considers Java-based application server platforms and related J2EE technologies. Because the focus of VIDE are business application involving databases, a data persistence framework base on JPA (Java Persistence API) is addressed by the mapping. The other platform – ODRA (Object Database for Rapid Application development), presented in detail later in this report, is intended to allow investigating the opportunities and limitations of code generation for a more homogenous, object-oriented environment. It also serves for supporting the research on extending OMG specifications towards the area of object-oriented database management systems. To this extent, unification between UML semantics and ODRA's underlying Stack Based Architecture is attempted. The mappings for both platforms also cover the Web service based connectivity, so that the abstract services being specified in VIDE can have their direct, executable, but also fairly platform-independent counterparts.

Since the source form for the model transformations specified in this document is VIDE metamodel, the relevant parts of the metamodel defined in [VIDE2007c] are summarized here mainly through UML static diagrams and followed with respective transformation rules. The transformations are specified in a generic way, however, where necessary, they are additionally illustrated with an example.

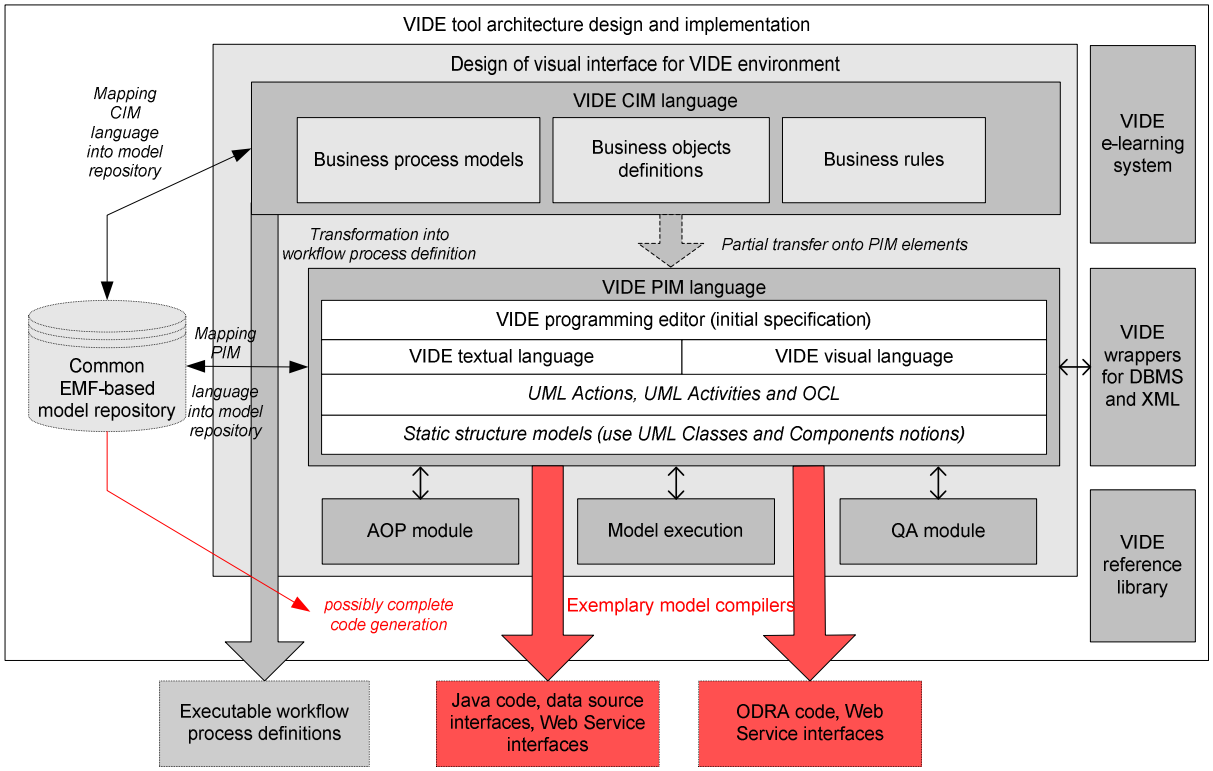


Figure 1 : Deliverable D6.1 in the overall project context

Work package 6 constitutes the latest step of the research phase of VIDE project. As shown in Figure 2, it directly interfaces with the following work packages:

- WP1 – by addressing the assumptions and requirements specified in that work package, in the area covered by WP6. The description of how those requirements are addressed can be found in Chapter 2 of this document.
- WP2 – by defining the VIDE PIM-level lanuage together with its metamodel, which consititutes the source for the mappings designed in the course of WP6. Some design decisions behind the VIDE PIM language are briefly explained in Chapter 4.
- WP8 and WP9 – these work packages, being performed to some extent parallel to each other and iteratively, set the actual realization of the project's research results, including the transformations defined in WP6. This work package defines the transformation to be used by the components: Java Model Compiler, ODRA Model Compiler and Model Execution Engine, specified in WP8 and being implemented in the course of WP9.

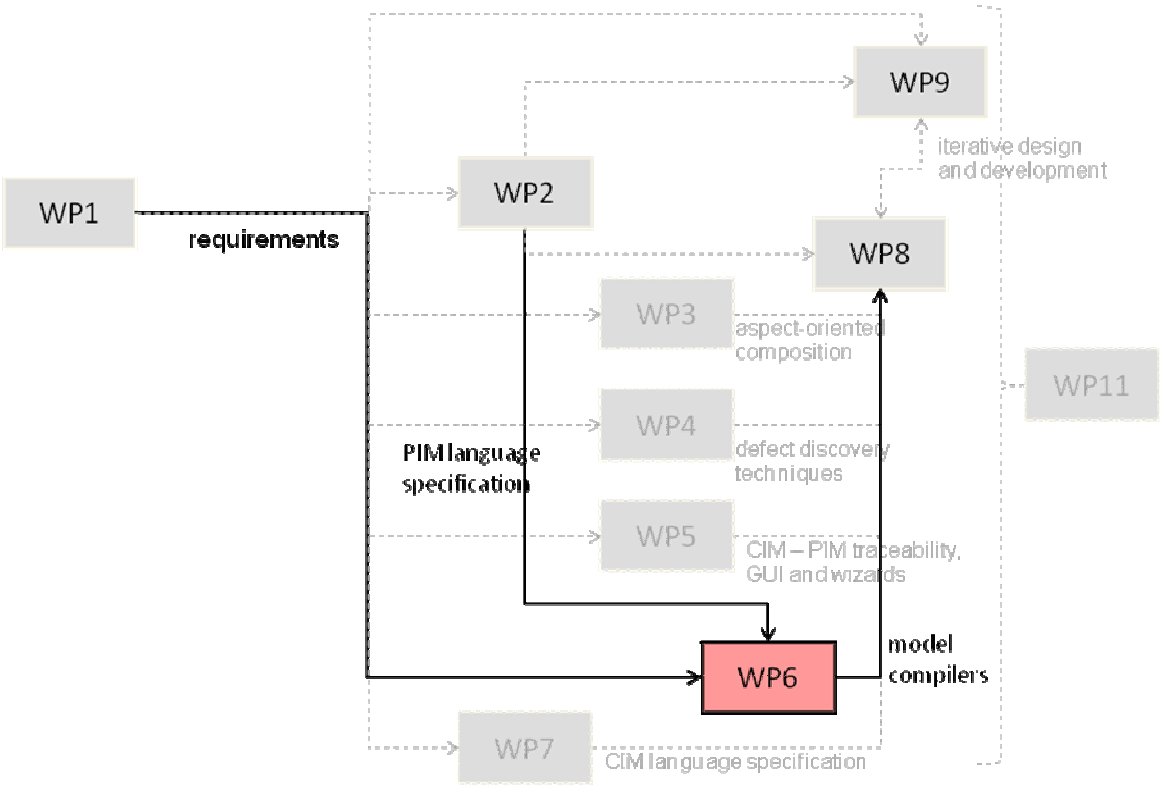


Figure 2 : Work package 6 in the overall project work flow

2 Requirement refinement

We provide here a list of requirements with respect to the VIDE project, collected in the deliverable document D1.1 [VIDE2007b] (see that document for a detailed description of these requirements) and indicate those found relevant for the WP6 scope. In the column "comment" we provide the relation of each requirement to the VIDE language, which is the subject of this deliverable document. For clarification, we denote which topics are subject of other work packages. We also sketch how WP6 addresses the relevant goals.

Requirement Number	**Name**	**Priority**	**Comment**
REQ – NonFunc 1	Accessibility at the CIM level	Should	Outside D6.1 scope. Addressed by D7.1 and the CIM-to-PIM transition support functionality to be described in D5.1.
REQ – NonFunc 2	CIM level collaboration	May	Outside D6.1 scope. Supporting this requirement will be considered in the course of D9.3 development.
REQ – NonFunc 3	On-line support for CIM/PIM users	Should	Outside D6.1 scope. Addressed in D5.1 (in the area of CIM-PIM navigation.
REQ – NonFunc 4	Clear and unambiguous notation – VIDE should have clear, comprehensible and unambiguous semantic description suited to the users of the VIDE tools	Should	Outside D6.1 scope. Addressed in D2.1
REQ – NonFunc 5	Model view saliency – VIDE models views must be user-oriented.	Should	The compilers have a contribution to this requirement because they allow users to think and concentrate on a PIM view of their problem, VIDE model, without cluttering this view with PSM consideration that are automatically handled by the compilers.
REQ – NonFunc 6	Appropriate textual/graphical fidelity – VIDE must provide appropriate textual and graphical modalities for its users.	Should	Outside D6.1 scope. Addressed in D2.1. CIM-related issues are subject of WP7 and WP5.
REQ – NonFunc 7	Timely feedback and constraints	Should	Outside D6.1 scope. Supporting the work of multiple users on a common model will be considered in the course of D8.1 and D9.1 development.
REQ – NonFunc 8	Runnable and testable VIDE prototypes	Should	D6.1, by defining mappings towards the executable platforms, lays a foundation for a systematic realization of this functionality. This requirement is more directly addressed in D9.0 and to be further investigated for D9.3.
REQ – NonFunc 9	Scalability of proposed solution – the proposed solution must at least conceptually scale to enterprise level.	Must	Regarding the large amounts of data, the use of JPQL queries and the support of generation for Web Services allow such scalability. J2EE system architecture for the Java compiler ensures enterprise scalability.
REQ – User 1	Flexibility and interoperability of VIDE language and tools - The VIDE language and tools SHOULD have	Should	The choice of OpenArchitectureWare as the framework of the Java compiler implementation, mainly because of its integration within Eclipse contributes to this requirement.

	flexibility and be interoperable with some existing tools.		
REQ – User 2	Reuse of UML Standard – end users are very sensitive to using standards. A key aspect is that the VIDE language reuses as much as possible the UML standard.	Should	Not strictly inside the scope of D6.1. Nevertheless, it should be noted that as the input metamodel to the compilers is UML and OCL based, it ease the understanding of compilers transformation.
REQ – Semantics 1	Semantics of VIDE Inte rnal Communication – a precise description of the semantics is needed sufficient for internal communication purposes within implementation stakeholders in the development of the VIDE tool.	Should	Thanks to the UML and OCL based metamodel, a clear semantic (although some interpretation variants are possible) is available that allows the definition of transformation rules to Java and ODRA.
REQ – Semantics 2	Simple VIDE semantics – after a first analysis it seems sufficient that the semantics of VIDE is described in natural language.	Should	No restriction on the metamodel has been found during the study of the mappings.
REQ – Lang 1	Usage of UML2 Behaviour ("Action Semantics") – VIDE should use the behavioural model elements of UML2 (earlier known as "UML Action Semantics"), unless proven insufficient.	Should	No restriction on the metamodel has been found during the study of the mappings.
REQ – Lang 2	Simplified UML meta-model – If it turns out that • the UML meta-model is unnecessarily complex in a way that it blocks the creation of a sensible concrete syntax (see remarks on ConditionalNode), • not all of the UML meta-model can be covered	May	No restriction on the metamodel has been found during the study of the mappings. Nevertheless some complexity, inherited from the UML metamodel, remains in the VIDE metamodel and leads to propose some modifications. See section 10.

	• elements are missing which are located in another needed language (like OCL) it may be changed.		
REQ – Lang 3	User Language & Concepts – the VIDE language and VIDE tools presented to a certain user groups SHOULD employ the language that is understood by the user group.	Should	Outside D6.1 scope.
REQ – Lang 4	Compliance with Standards – VIDE should not compete with existing adopted modelling standards, especially those adopted by the OMG, such as UML or BPMN.	Should	One of the 2 compilers proposed translate from VIDE to Java, a well known programming language widely used in the industry and well defined.
REQ – Lang 5	Deviation from Standards – VIDE may deviate in parts from existing standards, if a standard-conformant way is provided as well and if there are good reasons with respect to the overall user requirements.	May	No deviation from existing standards was made in D2.1.
REQ – Lang 6	Modularisation and extensibility – it should be possible to replace parts of the language with different artefacts and add additional language constructs for special business specific patterns. This requires the language to be structured in modules.	Should	Outside D6.1 scope.
REQ – Lang 7	Language for CIM, PIM, PSM modelling: 1) VIDE SHOULD support requirements definition tasks and business process description with BPML 2) VIDE SHOULD adopt action semantics for the modelling of	Should	Ad. 1. Outside 62.1 scope. Addressed in D7.1. Ad. 2. Outside 62.1 scope. Addressed in D2.1. Ad. 3. Compilation to Java is described in D6.1, compilation to other mentioned language is also possible without restriction.

	executable PIM models 3) VIDE SHOULD provide support for target PSM environments e.g. Java, C++, or SmallTalk; VIDE should provide platform implementation mappings in PIMs or CIMs.		
REQ – Tool 1	Usage of industrially adopted tools – VIDE must use industrially adopted meta-modelling standards where applicable.	Must	The compilers are integrated with Eclipse platform.
REQ – Tool 2	Meta-modelling Framework – VIDE must use EMF as its modelling framework.	Must	The compilers are defined on top of EMF modelling framework. Moreover OpenArchitectureWare, the framework selected for implementing the Java compiler is based on EMF.
REQ – Tool 3	Meta-modelling Concepts – VIDE meta-models should be constructed to be compatible with MOF concepts.	Should	Outside D6.1 scope. Addressed by D2.1.
REQ – Tool 4	M2M Transformation Technology (VIDE should use ATL as it's transformation framework, unless it is proven insufficient)	Should	This technology has not been used because of the lack of operational metamodel and code mappings for Java and ODRA
REQ – Tool 5	M2T Transformation Technology (VIDE should use XPAND as its M2T transformation language, unless proven insufficient.)	Should	This technology has been widely used for both compilers.
REQ – Tool 6	T2M Transformation Technology (VIDE should use XText framework, unless proven insufficient. An alternative can be parsers generated with ANTLR or LPG.)	Should	Outside D6.1 scope. To be addressed in D9.3.
REQ – Tool 7	Meta-modelling Framework (VIDE SHOULD use GMF as it's graphical modelling framework)	Should	Outside D6.1 scope. To be addressed in D9.1 and D9.3.

REQ – Tool 8	Use of OCL – VIDE should re-use existing standards as UML (REQ – User 1), and in particular OC; the goal is to achieve a seamless integration with the concrete syntax of the action language to be developed.	Should	Outside D6.1 scope. Addressed in D2.1.
REQ – Tool 9	CIM modelling standards.	May	Outside D6.1 scope. Addressed in D7.1.
REQ – Tool 10	PIM, PSM modelling standards – VIDE SHOULD provide support for PIM modelling with UML and action semantics; the meta-modelling standard for VIDE should be Ecore. VIDE SHOULD support well known PSM modelling standards (e.g. XMI for model and meta-model interchange, JMI for Java based PSM).	Should	Outside D6.1 scope. Addressed in D2.1.
REQ – Tool 11	Framework for CIM, PIM, PSM modelling	Should	The transformation technology for the Java compiler adopt the M2T paradigm. See section 9 for more details.
REQ – Tool 12	VIDE extensibility	Should	Outside D6.1 scope. To be addressed by D9.3.
REQ – Tool 13	Integration and metadata interchange – VIDE should provide model and meta-data interchange capability by adopting the XMI standard.	Should	Outside D6.1 scope. Addressed by D2.1
REQ – Tool 14	Model driven approach The VIDE tool strictly follows a model driven approach as stipulated in figure 9 page 120 of the D.1.1 deliverable	Must	The compilers bring their contribution to this requirement because they permit the transformation from PIM level in VIDE language to PSM level, either Java or ODRA.

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

3 Source Model

3.1 Global view

The source model from which translation to Java occurs is the VIDE metamodel defined in [VIDE2007c]. It is briefly presented hereafter. Figure 3 shows a high level view of the VIDE metamodel. While technically not structured with these packages, this decomposition is useful to structure the specification of the mapping to Java.

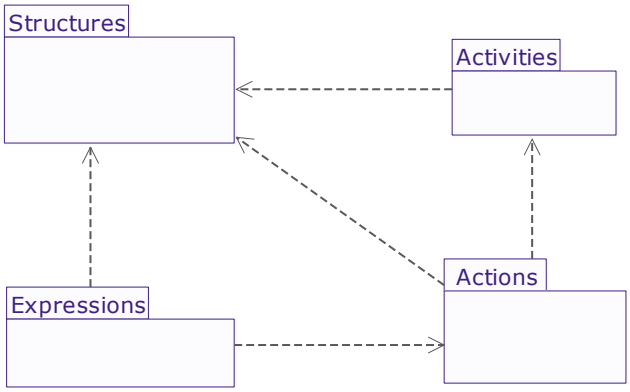


Figure 3 : VIDE metamodel main packages

3.2 Structures

This part describes the static structures (package, class, etc) of the VIDE metamodel as well as the base types and several high level classes derived in other packages.

Figure 4 presents the static class model of VIDE. Classes, which are types, belong to Packages and have Attributes (named Properties in the metamodel), and Operations, which have Parameters. Classes have inheritance relationship (Subclass, Superclass).

Association have two Property whose Type (not shown in the Figure) holds the linked classes.

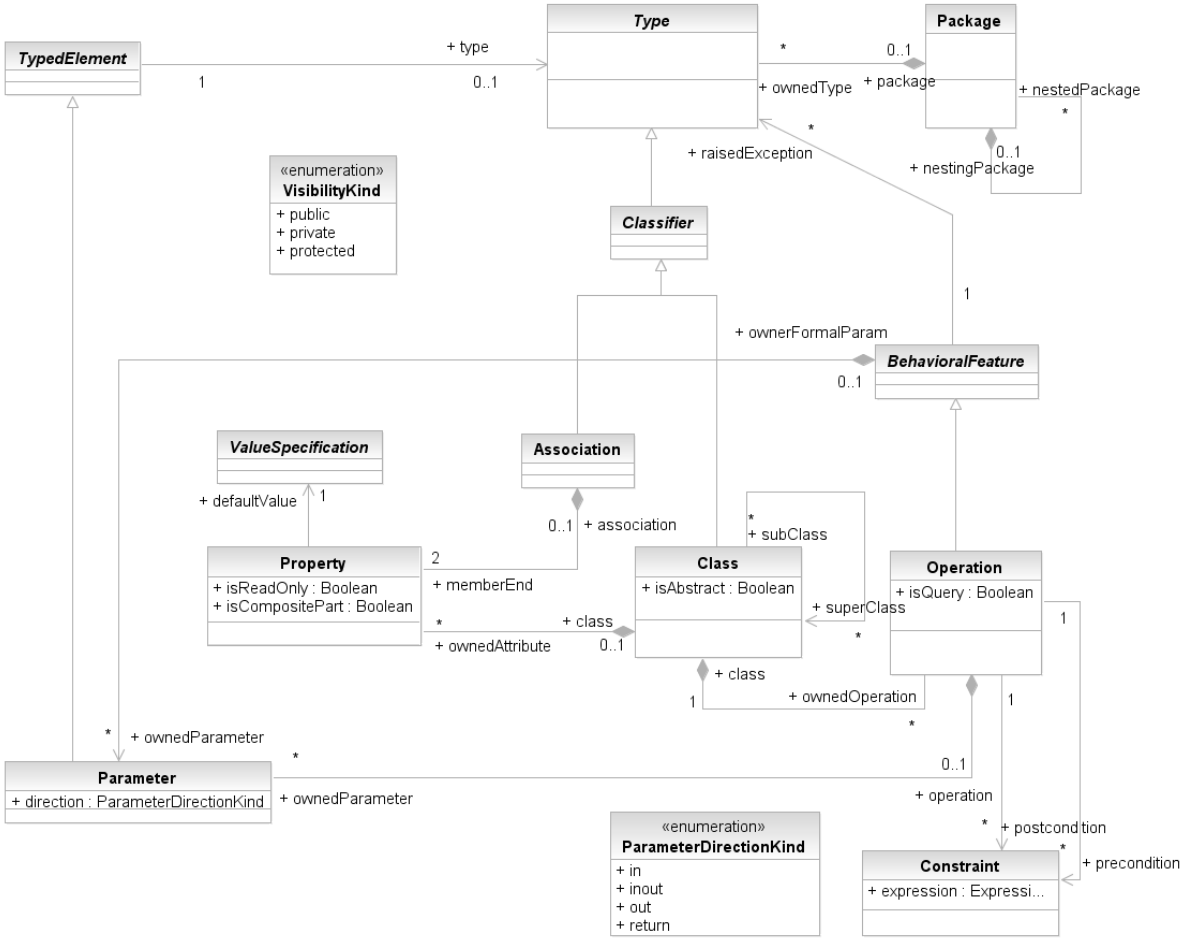


Figure 4 : VIDE class model

Figure 5 presents the Feature metaclass, which is derived into two main branches, BehavioralFeature from which Operation inherits and StructuralFeature to describes Property (attributes and association end).

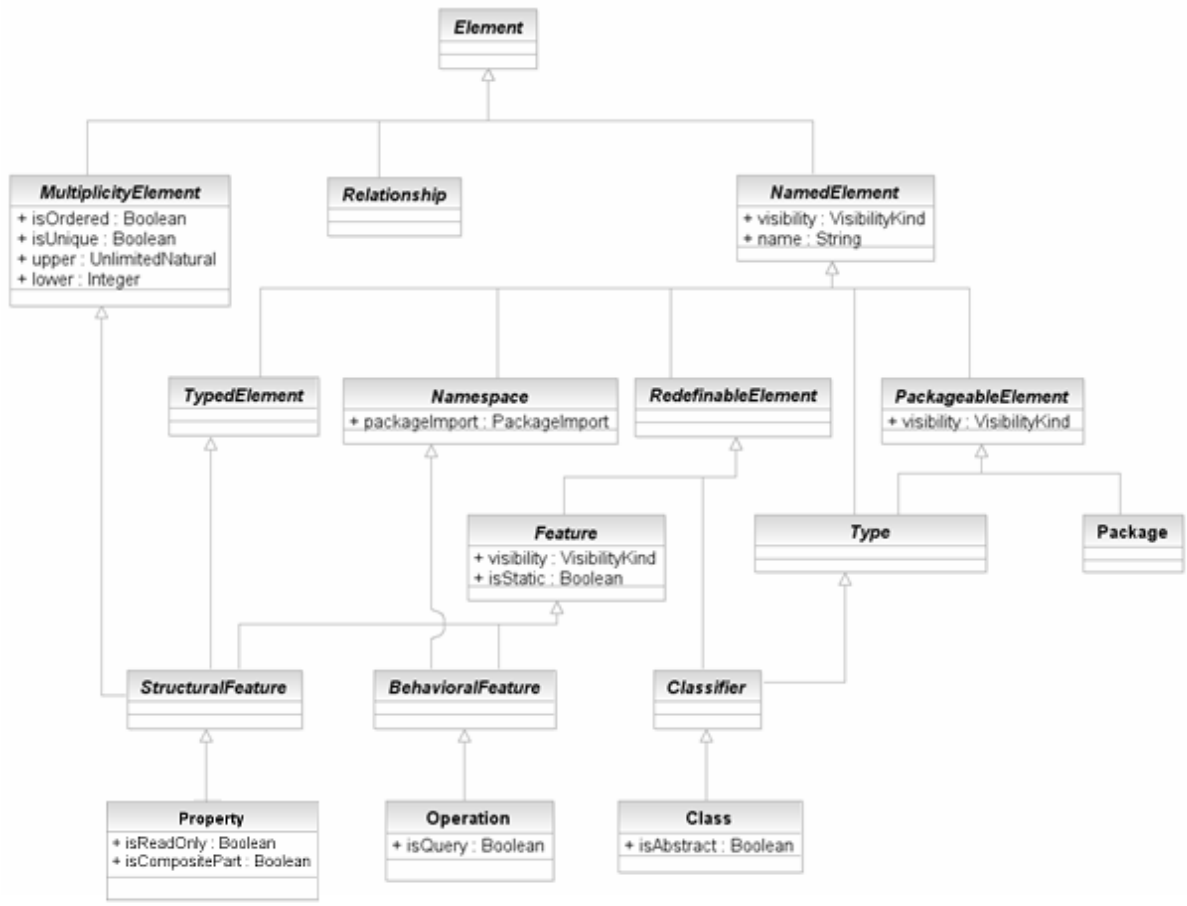


Figure 5 : VIDE Feature Model

Figure 6 presents the Package classes. It contains Type, it can be nested (nestedPackage) and any Namespace (Package, Class) can import packages.

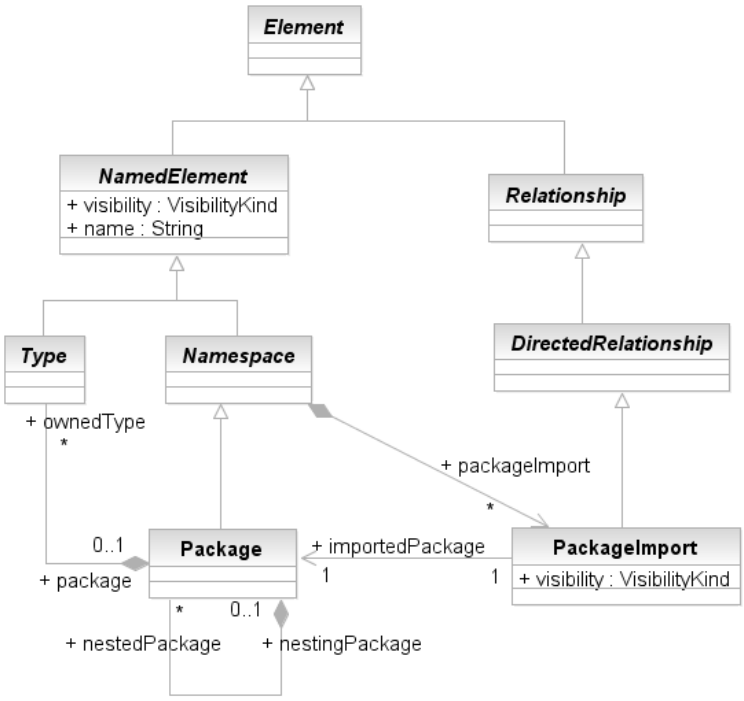


Figure 6 : VIDE Package and Import

Figure 7 presents the Type hierarchy. Beside Class and Association, DataType is the root of a rich family of type with atomic ones like Enumeration, Primitive and VoidType and aggregate ones with Tuple and heir of CollectionType. TupleType, VoidType and the heir of CollectionType all come from the OCL metamodel and are modelled as instances of DataType, at M1 level in the categories of model defined by OMG. This is presented Figure 8.

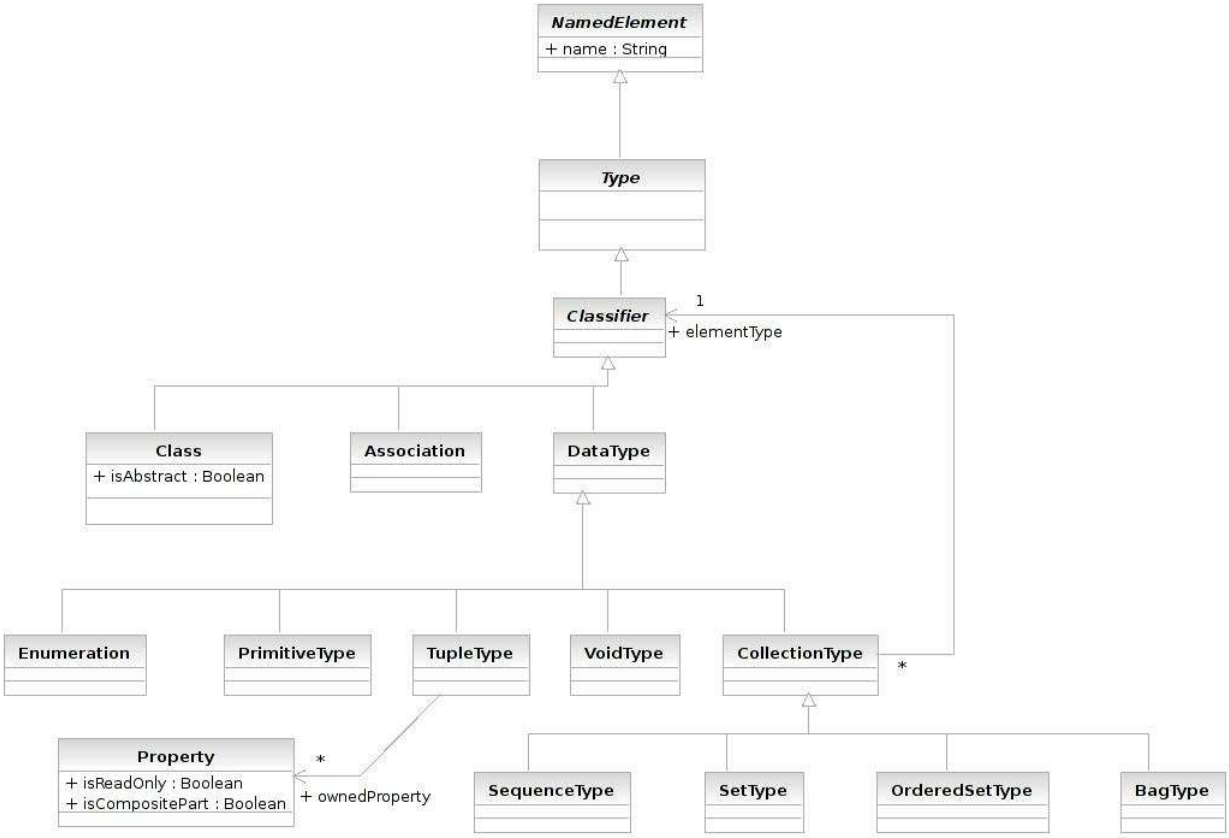


Figure 7 : VIDE Type hierarchy

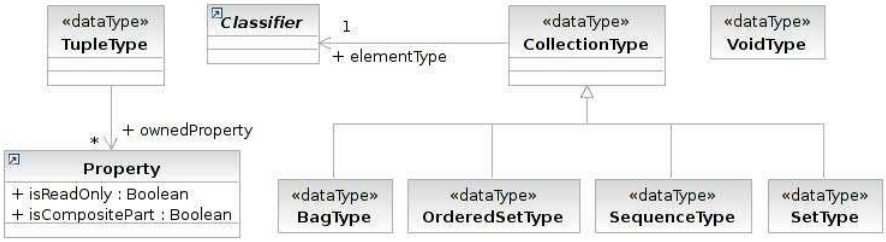


Figure 8 : VIDE Datatypes

3.3 Activities

This part describes activities. Activity is the element where actions and expressions are defined. It provides a context for the execution of these elements as well as a mean of ordering their sequences.

Figure 9 shows how Activity is connected to Operation in VIDE metamodel. The property method defined on the association between abstract classes BehavioralFeature and Behaviour allows navigating from an Operation to its Activity.

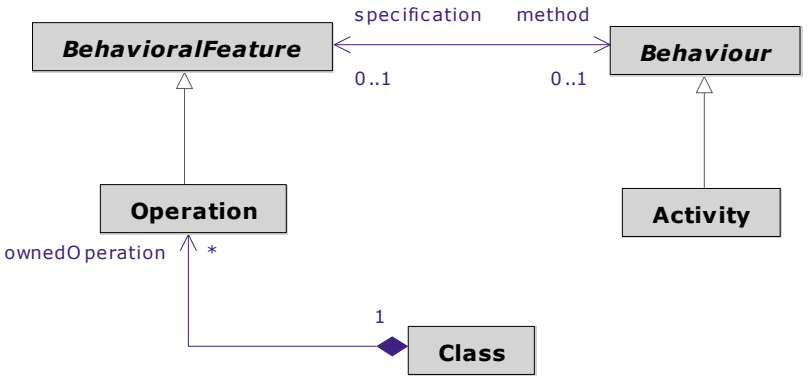


Figure 9 : Relationship between Operation and Activity in VIDE Metamodel

Figure 10 presents the metamodel of Activity. An Activity is composed of several ActivityNode, which can be ControlNode, ObjectNode or ExecutableNode. All these nodes are surrounded by ActivityEdge to define the sequence of execution. ExecutableNode is further refined in Action, the base class for all actions metaclasses, presented thereafter, and several nodes that allow a finer control of the execution flow: ExpansionRegion, ConditionalNode, LoopNode and SequenceNode.

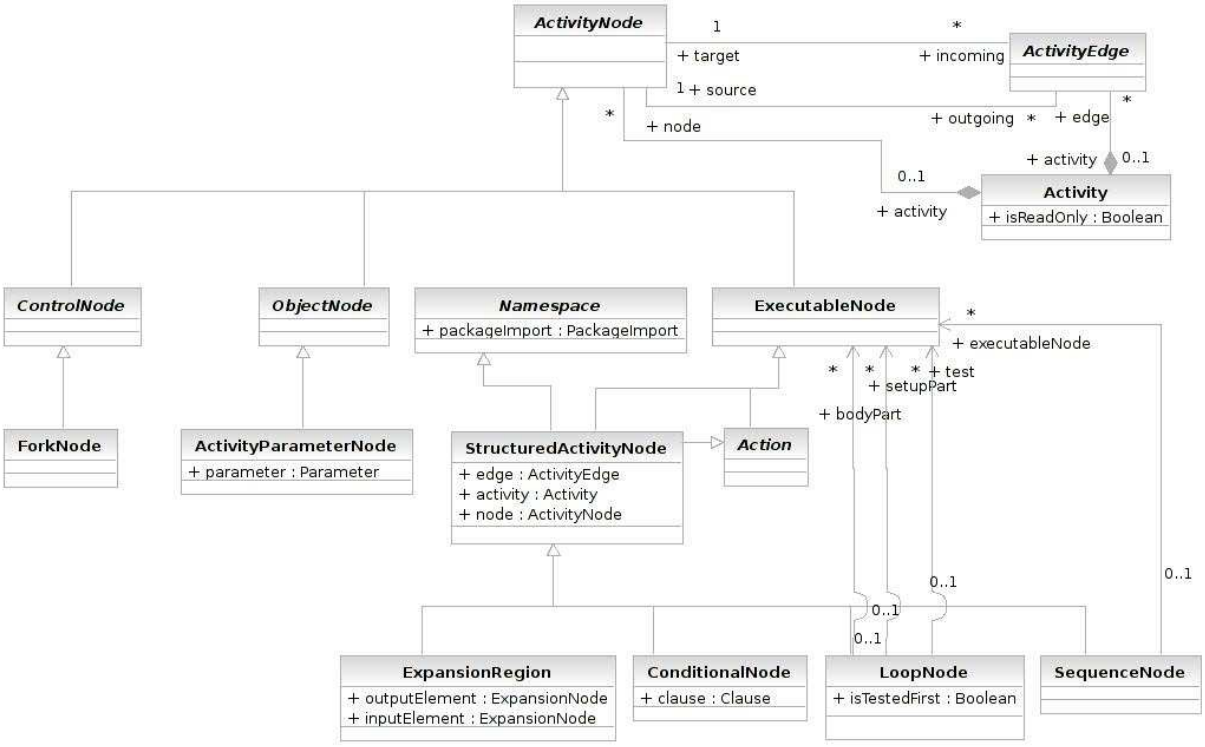


Figure 10 : VIDE Activity

Figure 11 presents the detail of the ConditionalNode, an important node to represent choice and alternative in algorithms. A ConditionalNode has one or more Clause, each having two ExecutableNode, one for the test (the ExecutableNode should returns a Boolean value) and

one for the body, which is executed if the test is true. Otherwise the next clause in the ordered clause association is executed.

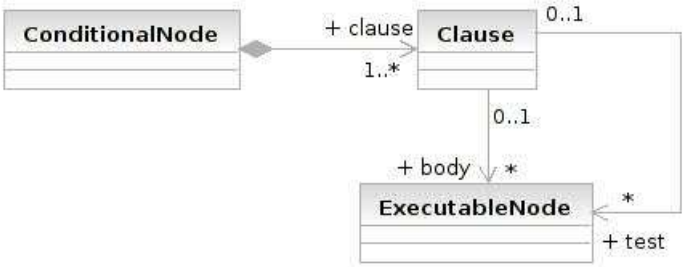


Figure 11 : Detail of ConditionalNode

Figure 12 details the metamodel for representing ExceptionHandler. ExeceptionHandler protect an ExecutableNode (association protectedNode) (which is also the owner of the ExceptionHandler, not shown in the figure). It has an exceptionInput (a parameter) which is an ObjectNode that should conform to the exceptionType, a Classifier. If the handler is triggered, it executes its handlerBody, an instance of ExecutableNode.

Figure 12 : Exception Handler

3.4 Actions

An action is the fundamental unit of executable functionality. The execution of an action represents some transformation or processing in the modelled system.

Figure 13 presents the detail of an action. An Action can have several input and output pins which can be seen as data consumed and produced by the action. It is important to note that InputPin can be associated with a ValueSpecification that describes the content of the InputPin and that description can be an OCL expression. An Action is executed inside a context as shown by the association to Classifier. This context will be an Activity.

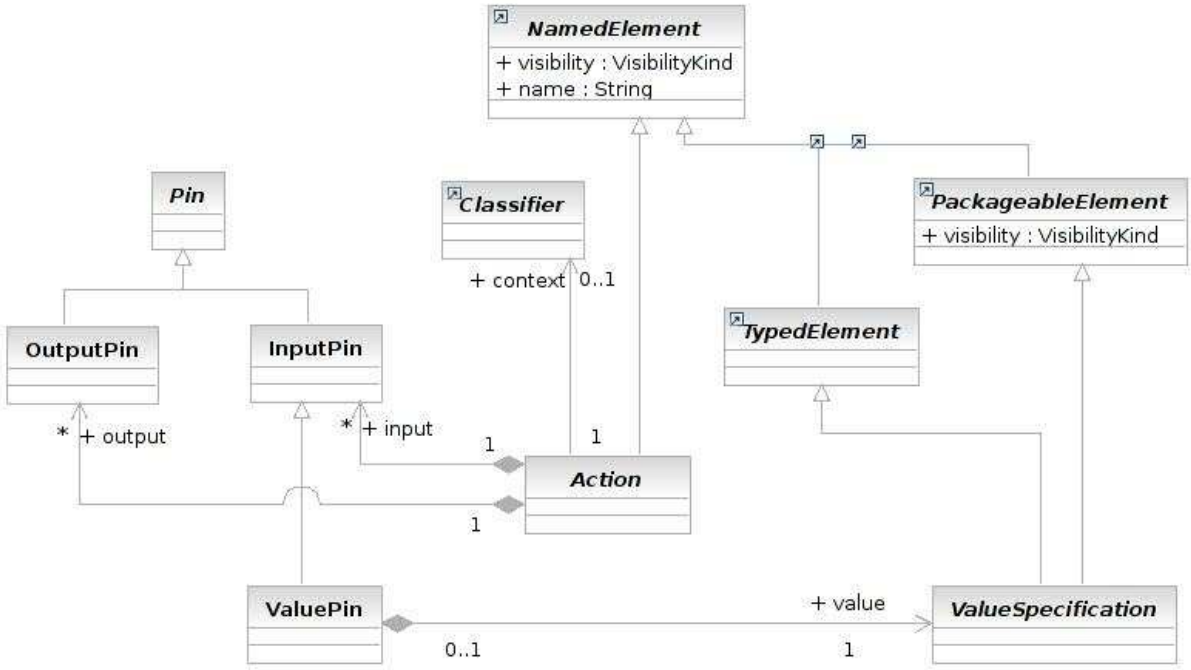


Figure 13 : VIDE Action metamodel

Figure 14 presents *CallOperationAction* and related actions. This action is useful to call an operation in an *Activity*. *ReplyAction* triggers the return of the current *Operation* and is able to return optionally multiple values. *RaiseExceptionAction* is used to trigger an exception. All these actions rely on input and output port to retrieve and produce values.

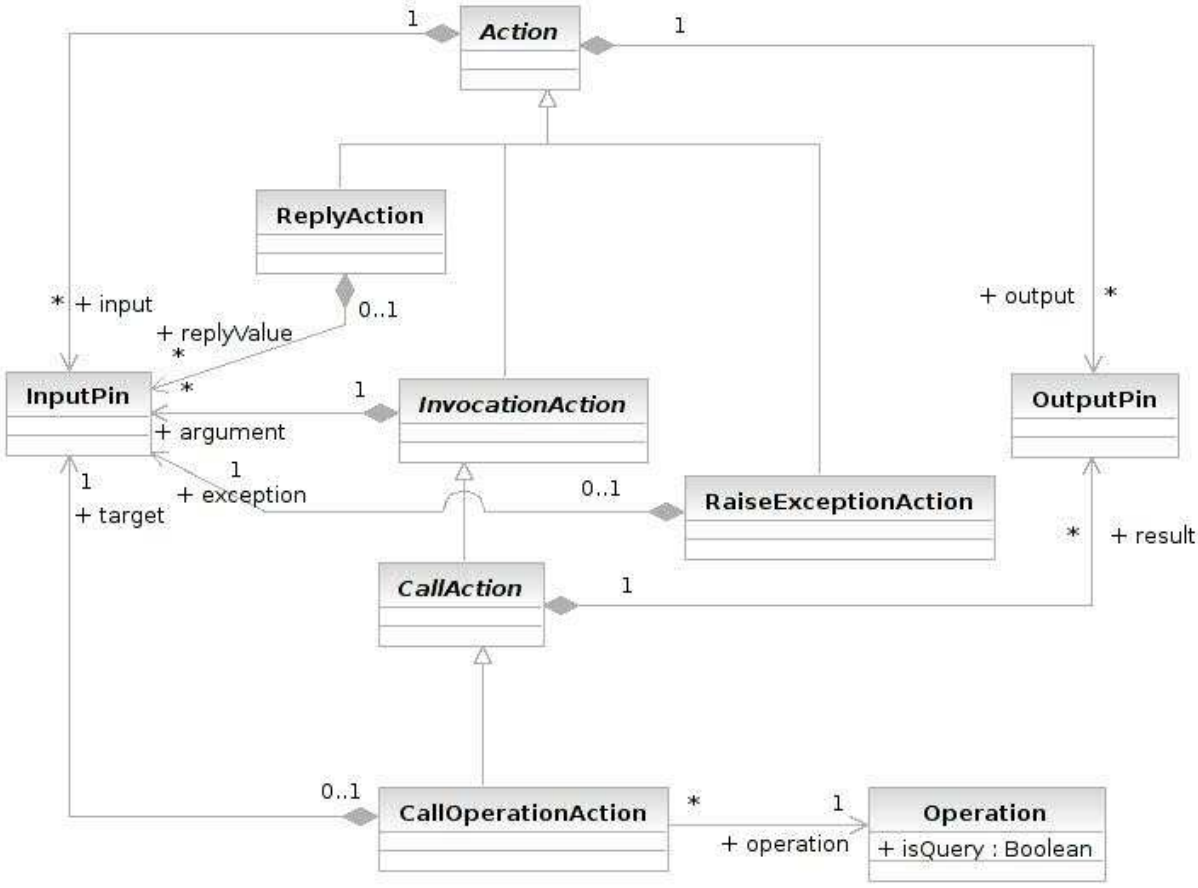


Figure 14 : Invocation Actions in VIDE Metamodel

Figure 15 presents the actions that create and delete objects.

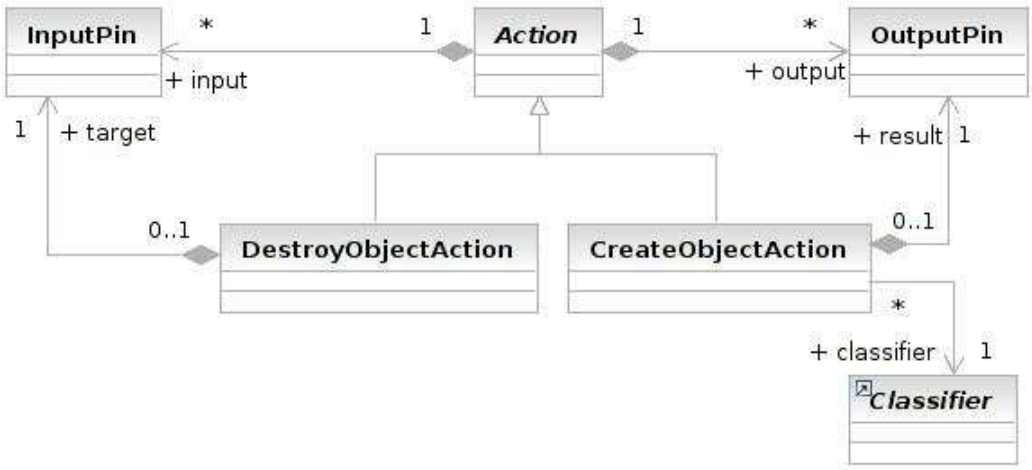


Figure 15 : Object Actions in VIDE Metamodel

Figure 16 presents actions that allow representing manipulation of properties (class attributes) generalized as *StructuralFeature*.

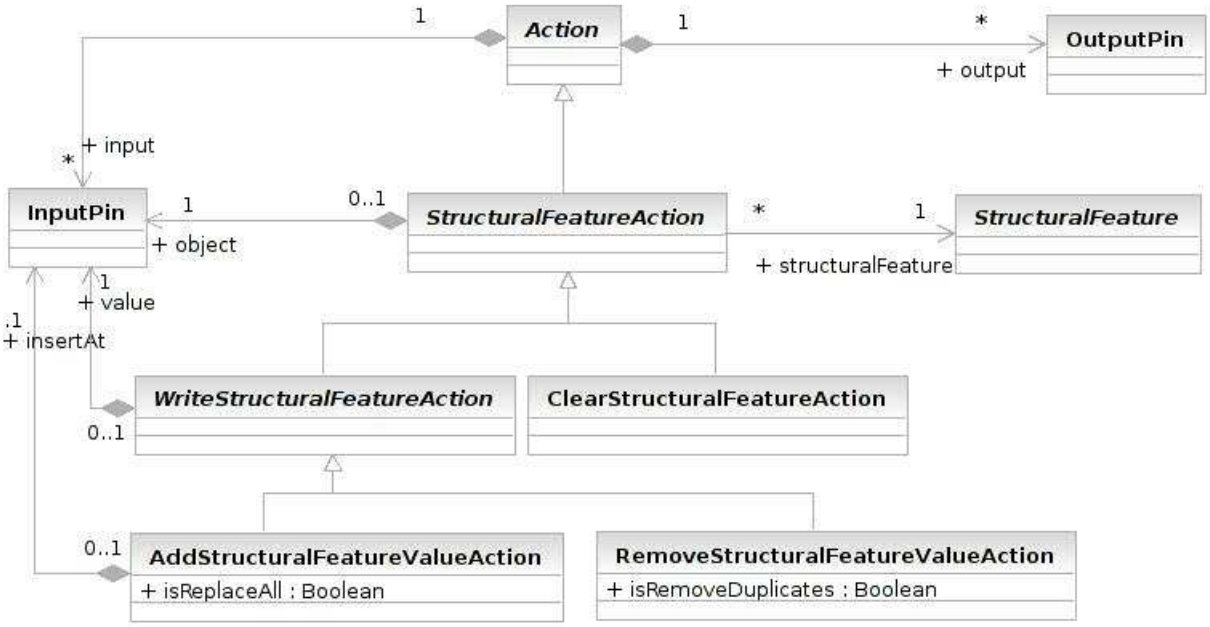


Figure 16 : Structural features actions in VIDE Metamodel

Figure 17 presents actions that allow representing manipulation of *Association*.

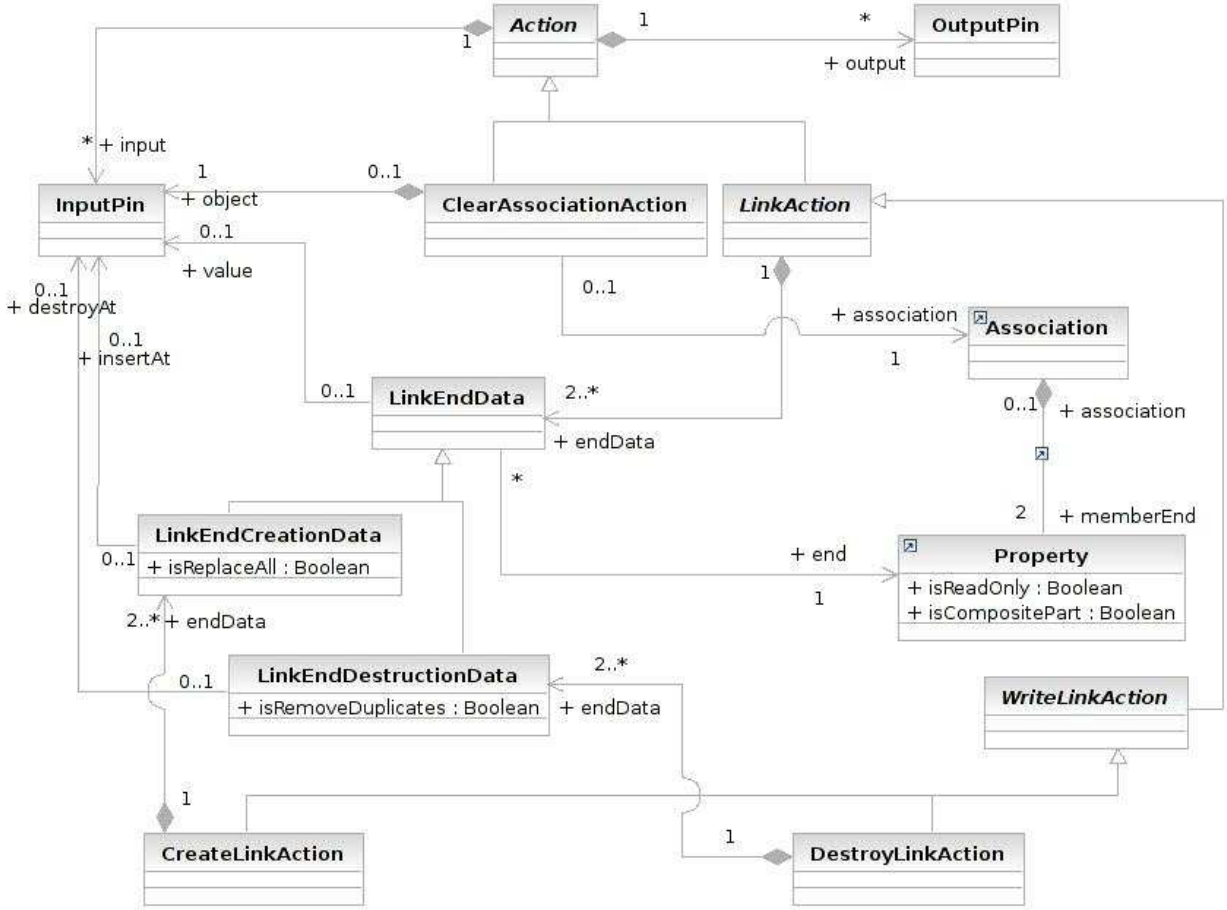


Figure 17 : Link Actions in VIDE Metamodel

Figure 18 presents actions that allow representing manipulation of *Variable* and *ValueSpecification.*

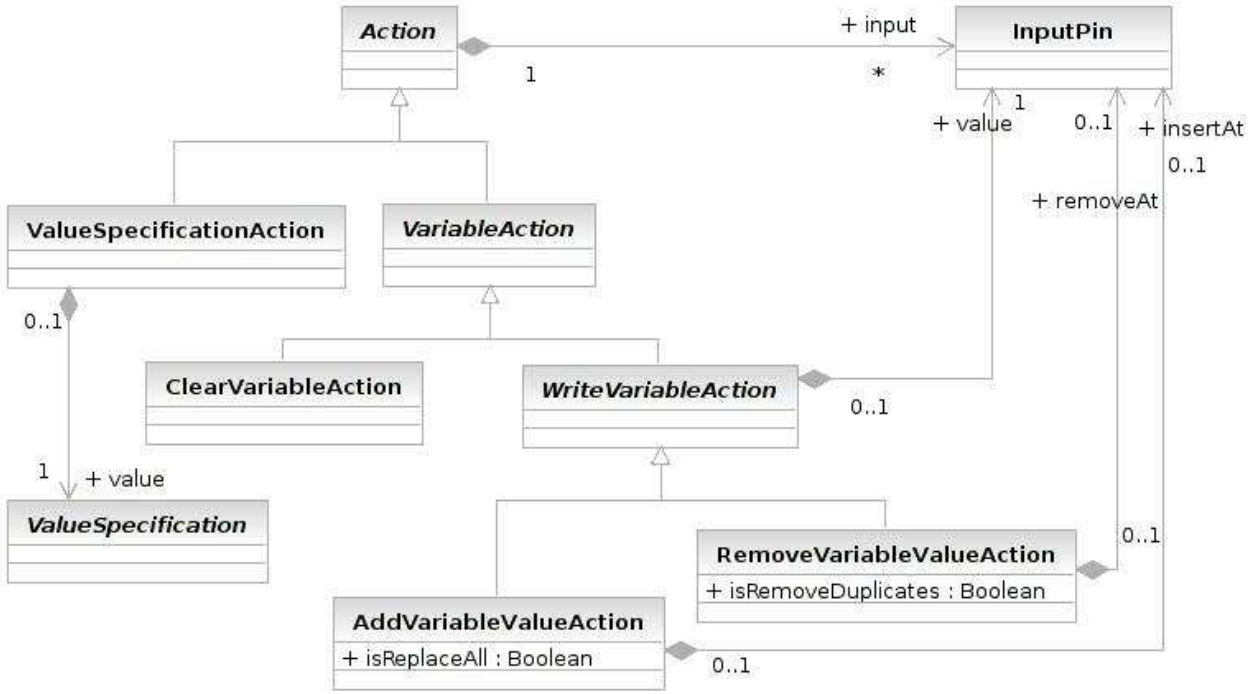


Figure 18 : Value and Variable Actions in VIDE Metamodel

3.5 Expressions

Expressions in VIDE are OCL expressions that can appear at any place where the metaclass ValueSpecification appears in the VIDE Metamodel.

Figure 19 presents the hierarchy of OCL expression as well as the connection with *ValueSpecification*. *ExpressionInOcl* is the root of any OCL expression. It can have an OclVariable that acts as the *this* pseudo-variable.

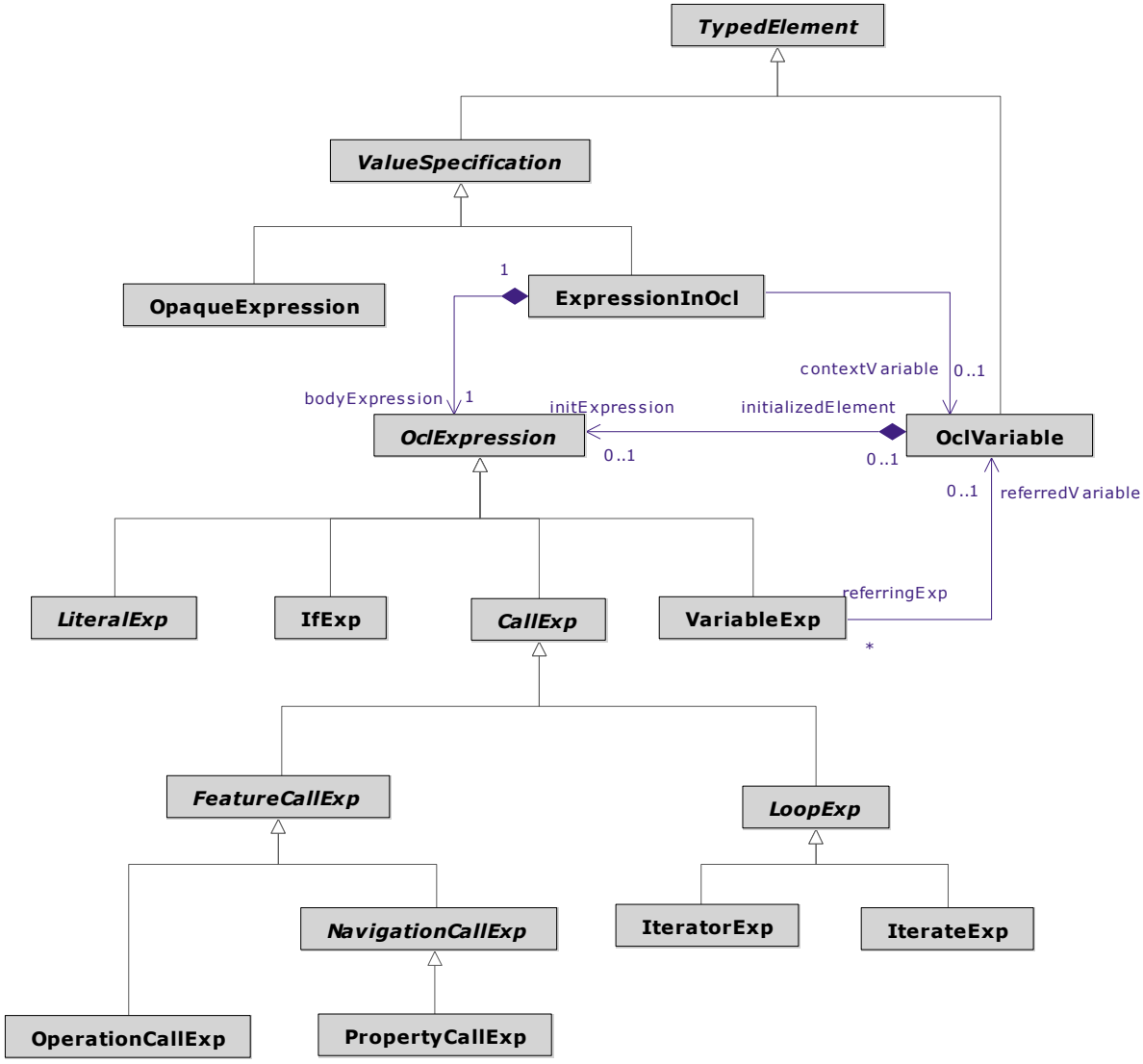


Figure 19 : OCL expressions and their connection to ValueSpecification in VIDE metamodel

Figure 20 presents the detail of the *LiteralExp* OCL expression.

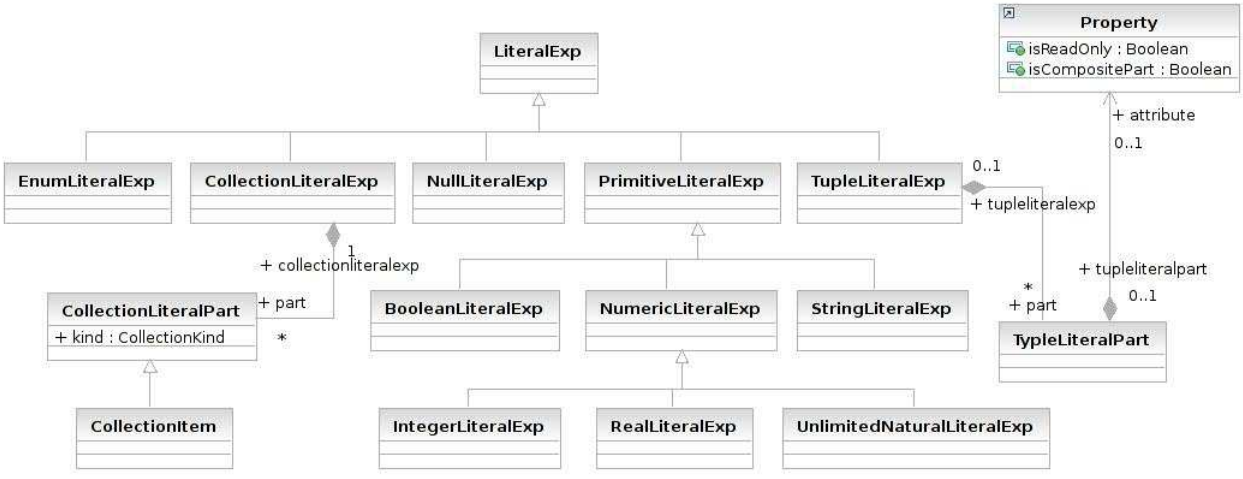


Figure 20 : Literal Expression in VIDE metamodel

Figure 21 presents the conditional and loop OCL statements. The one to one multiplicity of *elseExpression* property between *IfExp* and *OclExpression* indicates that the else clause of a well formed OCL expression is required.

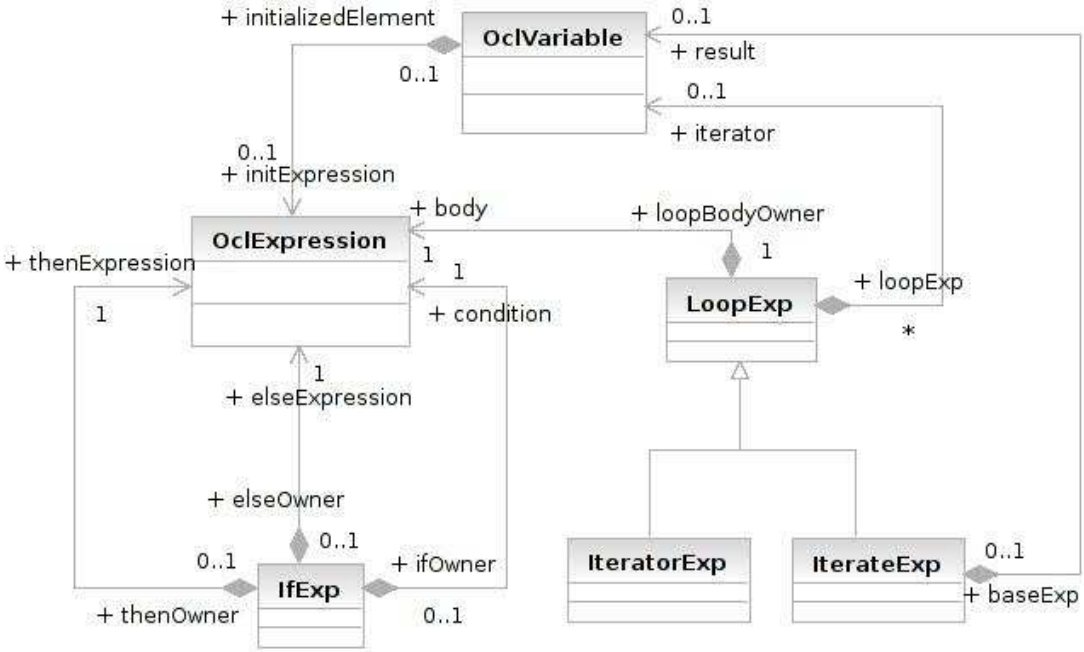


Figure 21 : Conditional and Iterator expression in VIDE metamodel

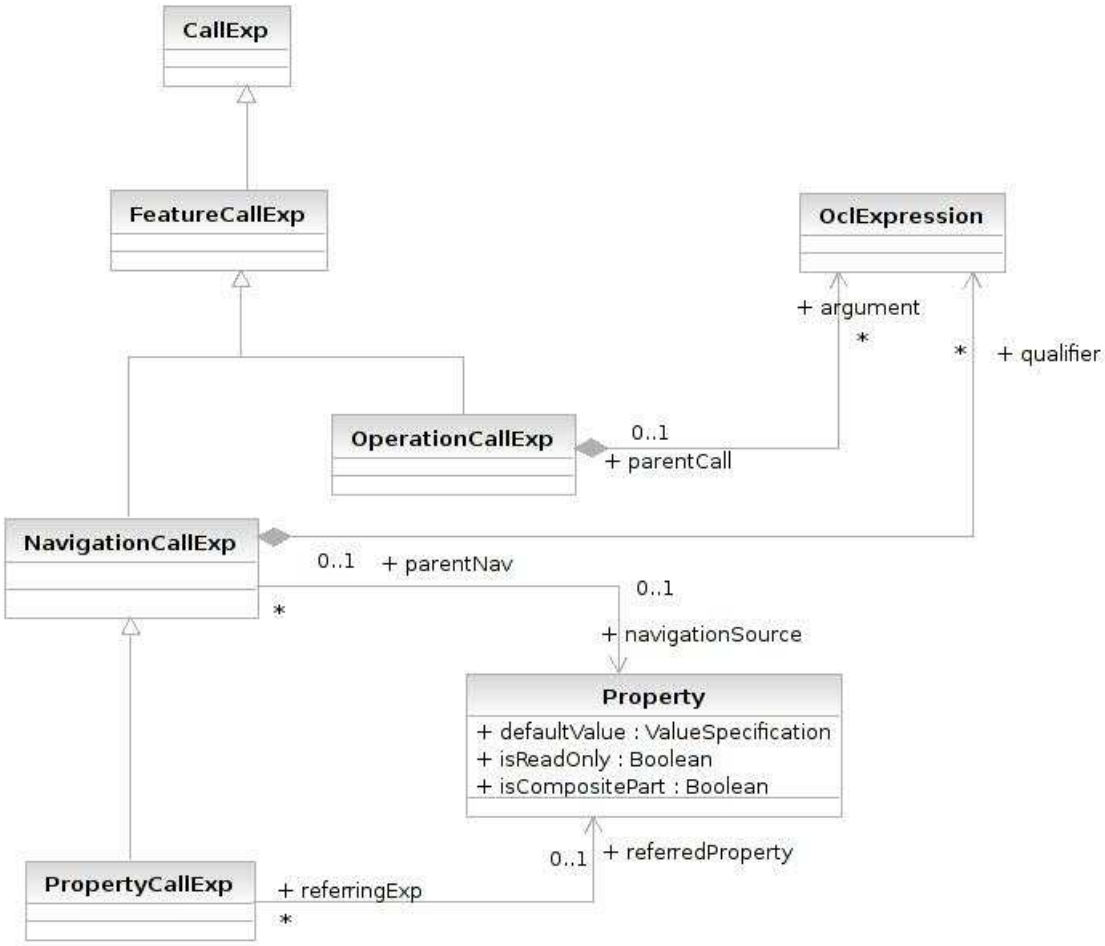


Figure 22 : OCL Operation call in VIDE Metamodel

4 Choices behind the VIDE metamodel design and query language selection

VIDE PIM level language used to express source models for the model compilers specified here consists of UML 2.1 and OCL 2.0 subsets. As described in [VIDE2007c], the main elements and their responsibilities are as follows:

- UML Static Structure (Classes, Packages) for specifying the model structure and data schema,
- UML Actions unit for representing atomic steps of application behaviour,
- UML Structured Activities, for representing the control flow inside methods,
- OCL, to cover all kinds of expressions in VIDE models, including complex queries.

The choice of UML, including its action semantics, was made already at the stage of project proposal and results from consortia interest in contributing to existing modelling standards and investigating the actual potential of the MDA approach that is based on them.

The features of object-oriented design provided by UML allow for using various OO patterns in modelling with VIDE, and seem to provide an adequate level of abstraction from the point of view of subsequent transformation into popular OO programming language code.

While the features of UML behaviour to be used are not very mature, and the reuse of existing, ready model compilers cannot be assumed, two important motivating factors for its choice can be indicated:

- The popularity of UML structural modelling constructs – that can provide a well known, platform neutral object model for precise modelling. The constructs are familiar to developers and moreover, can be readily supported by existing UML modelling tools.
- The presence of standard-compliant UML and OCL model repository implementations and related infrastructure at the Eclipse platform. Thanks to it, the project results can be potentially reusable within the community. The repository format and modelling frameworks handling it, radically simplify not only the editor tools development, but especially – provide necessary means for model processing (within PIM level, e.g. for aspect-oriented composition, as well as in model compilation and execution – e.g. code generation).

With the above factors in mind, the remaining choice, regarding the expression language for VIDE was significantly constrained. Depending on the basic means provided by UML Actions for data read does not satisfy the needs of language expressiveness. Firstly, it would undermine modelling productivity when processing complex data structures; secondly – this would cause a problematic situation where the modelling constructs are at a lower level of abstraction compared to target platform language features available. In this case one of two approaches could be followed:

- Developing a query language for UML from scratch (in terms of specifying its semantics, metamodel, concrete syntax, library functions etc.). Note that a potential choice of using some existing query language (e.g. SQL, OQL or XML Query) would

involve similar effort – as it would need to be adapted to a different data model (UML) than originally assumed. The way of handling such language constructs inside model repository would also need to be addressed. It is important to note that following too close the solutions known from platform-specific solutions (e.g. query languages and persistency frameworks) would contradict the expected benefits of uniform, platform independent language. For example, using the object-relational mappings actually forces the developer to depend on SQL (that is, the means comparable in terms of expressiveness with SQL-92) and to be aware of all the relational database details – hence only some aspects of the infamous "impedance mismatch" problem are removed in that case.

- Using an existing UML-compliant expression language – namely – OMG OCL 2.0. This choice resolves the problems of data model compliance, metamodel definition (including several aspects of integration into UML), model repository implementation and concrete syntax. However, at the same time, one needs to face shortcomings of OCL serving as a query language (the role that was considered at least secondary during its design).

With additional consideration of the ease of adoption in the modelling community, the latter path was chosen. This resulted in several refinements of the original specification, in terms of its integration with UML behaviour, as well as a slight extension of the OCL standard library functions. Obviously, the resulting language cannot match in terms of overall maturity the existing industrial solutions originally designed as query languages. However, it offers analogous expressive power. A certain usability problems results from OCL syntax, which is rather complex and less friendly than other query languages. This is especially visible in a very complicated way a join expression (foundational for query languages) can be achieved in OCL. The visual solution for building OCL expression has been designed to relieve this problem.

5 Target Platforms

5.1 J2EE Reference Application

The Java 2 Platform, Enterprise Edition (J2EE) defines the standard for developing multitier enterprise applications. The J2EE platform simplifies enterprise applications by basing them on standardized, modular components, by providing a complete set of services to those components, and by handling many details of application behaviour automatically, without complex programming.

The J2EE platform takes advantage of many features of the Java 2 Platform, Standard Edition (J2SE), such as "Write Once, Run Anywhere" portability, JDBC API for database access, CORBA technology for interaction with existing enterprise resources, and a security model that protects data even in internet applications. Building on this base, the Java 2 Platform, Enterprise Edition adds full support for Enterprise JavaBeans components, Java Servlets API, JavaServer Pages and XML technology. The J2EE standard includes complete specifications and compliance tests to ensure portability of applications across the wide range of existing enterprise systems capable of supporting the J2EE platform. In addition, the J2EE specification now ensures Web services interoperability through support for the WS-I Basic Profile.

Figure 23 presents the latest version of Java Enterprise Architecture API in a typical multi-tiers application.

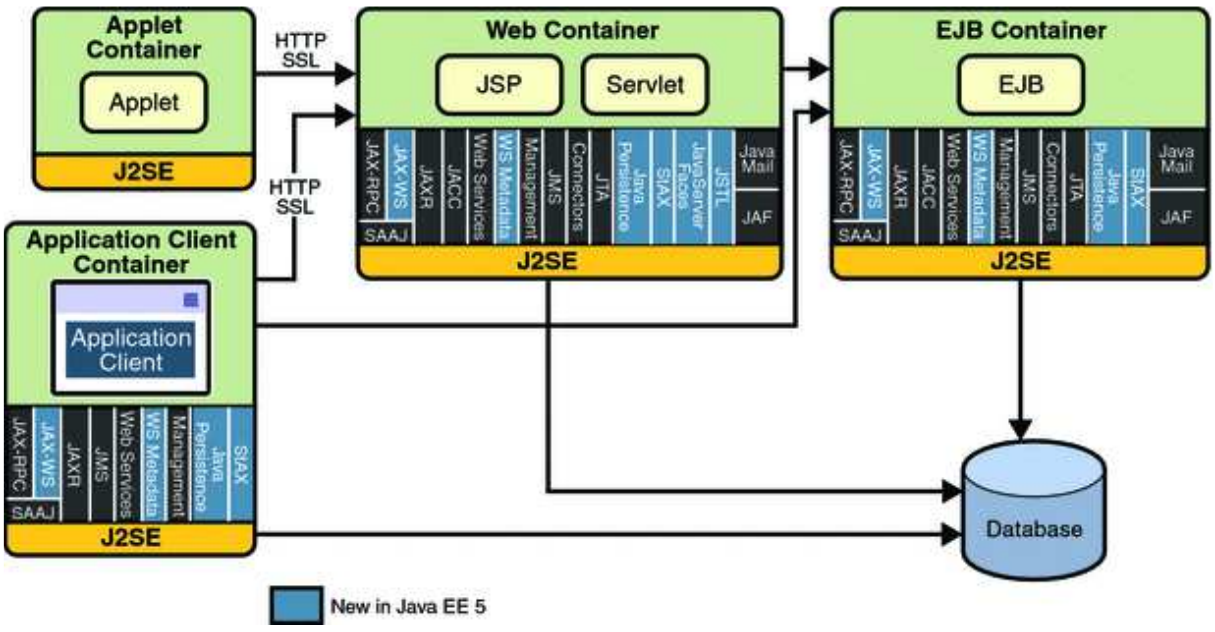


Figure 23 : Java EE 5 Architecture Overview

J2EE provides many possibilities for system architecture. Therefore, the target system architecture for the J2EE Compiler needs to be chosen. Since VIDE programs focus on **business/behavioural logic** that operate on a **database** and that are accessible via **Web**

Services the following APIs of the J2EE Platform have been considered for the Java Compiler:

1. Java Web Services (JAX-WS) for web services definition and consumption
2. EnterpriseJavaBean (EFJB) that containing the business logic
3. Java Persistence API (JPA) to access to RDBMS from VIDE programs.

These APIs have been selected because they offer strategic services for modern service oriented applications on typical information systems and because VIDE is conceptually designed to support these APIs.

5.2 SAP Application Server Variant

SAP NetWeaver Application Server [SAPAS] provides a complete infrastructure for developing, deploying, and running enterprise applications. SAP NetWeaver Application Server supports both Java technologies and ABAP. As it is based on industry standards, SAP Netweaver Application Server provides an open platform that allows an easy integration of applications and processes.

SAP NetWeaver Application Server 7.1 is a certified Java 5 Enterprise Edition application server. It supports the latest Java EE 5 features such as Java API for XML Web Services (JAX-WS 2.0), Java Persistence API (JPA 1.0), Enterprise JavaBeans (EJB 3.0), Java Server Faces (JSF 1.2), etc.

Since it is fully compliant with Java 5 EE, the proposed solutions for handling Web Services (based on JAX-WS 2.0) and persistence (based on JPA 1.0) in the Java compiler work seamlessly with SAP Netweaver Application Server. That is, the generated code with Java annotations for Web Services and persistence runs on any Java 5E EE compliant server including SAP Netweaver Application Server. However since SAP NetWeaver Application Server requires a dedicated packaging format (SDA – Software Delivery Archive), compilation of VIDE J2EE programs before their deployment is required.

5.3 ODRA

ODRA (Object Database for Rapid Application development) [ADHK2008] is an object-oriented application development environment currently being constructed at the Polish-Japanese Institute of Information Technology. The aim of the project is to design a next-generation development tool for future database application programmers. The tool is based on the query language SBQL (Stack-Based Query Language), a new, powerful and high level object-oriented programming language tightly coupled with query capabilities. The SBQL execution environment consists of a virtual machine, a main memory DBMS and an infrastructure supporting distributed computing. The main goal of the ODRA project is to develop new paradigms of database application development, by increasing the level of abstraction at which the programmer works. It introduced a new, universal, declarative programming language, together with its distributed, database-oriented and object-oriented execution environment. The intent is to provide functionality common to the variety of popular technologies (such as relational/object databases, several types of middleware, general purpose programming languages and their execution environments) in a single universal, easy to learn, interoperable and effective to use application programming environment.

ODRA consists of three closely integrated components:

- Object Database Management System (ODBMS)
- Compiler and interpreter for object-oriented query programming language SBQL
- Middleware with distributed communication facilities based on the distributed databases technologies.

The system is additionally equipped with a set of tools for integrating heterogeneous legacy data sources. The continuously extended toolset includes importers (filters) and/or wrappers to XML, RDF, relational data, web services, etc.

Fig.1 presents a view on the architecture, which involves data structures (figures with dashed lines) and program modules (grey boxes). The architecture takes into account the subdivision of the storage and processing between client and server, strong typing and query optimization (by rewriting and by indices). The subdivision on client and server is only for easier explanation; actually, each ODRA installation can work as a client and as a server. Many clients can be connected to a server and a client can be connected to many servers. Below we present a short description of architectural elements from Figure 24.

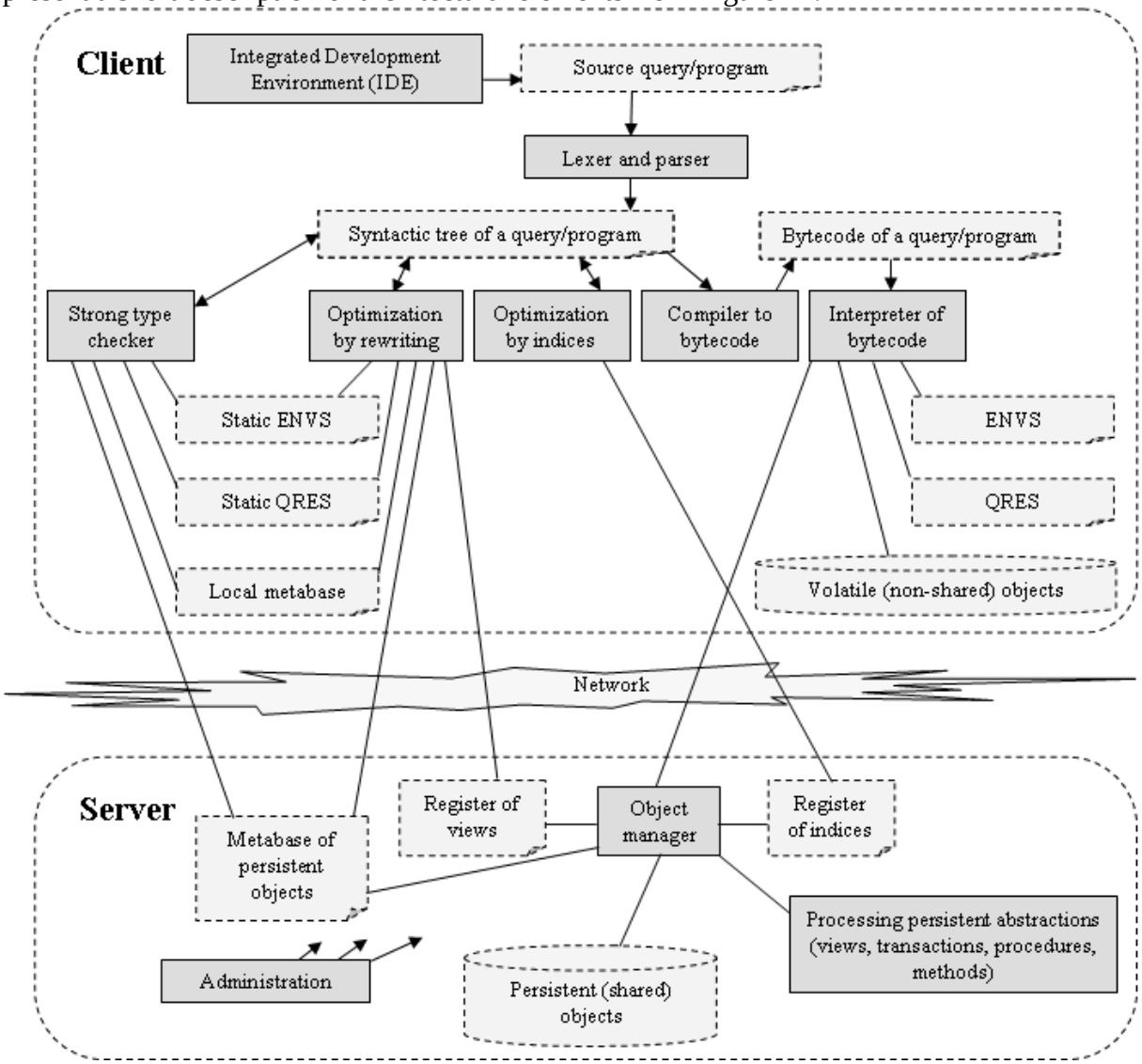


Figure 24 : Architecture of ODRA

In the figure above it is worth to note the following elements:

- The **strong type checker** takes a query/program syntactic tree and checks if it conforms to the declared types. Types are recorded within a client local metabase and within the metabase of persistent objects that is kept on the server. The strong static type checker simulates actual execution of a query during compile time. The type checker has several other functions. In particular, it changes the query syntactic tree by introducing new nodes for automatic dereferences, automatic coercions, for typing literals, for resolving elliptic queries and for dynamic type checks (if static checks are impossible). The type checker introduces additional information to the nodes of the query syntactic tree that is necessary further for query optimization.
- **Static ENVS** - static environment stack. It is a compile time counterpart of the environment stack (call stack) known from almost all programming languages.
- **Static QRES** - static result stack. It is a compile time counterpart of the result stack (arithmetic stack) known from almost all programming languages.
- **Optimization by rewriting** - this is a program module that changes the syntactic tree that is already annotated by the strong type checker.
- **Compiler to bytecode.** This module takes the strongly checked and optimized syntactic tree of a query/program and produces a bytecode that can be executed by the interpreter. In the prototype implementation we developed our own bytecode format called Juliet. In the future we consider the possibility to generate directly the Java bytecode but it needs further research.
- **Updateable object views.** ODRA offers a highly transparent mechanism for updateable object views that allows defining virtual objects with arbitrary, explicitly definable update semantics. This feature is essential for integrating various data sources using ODRA.

ODRA introduces a powerful query and programming language SBQL (Stack-Based Query Language). It is precise with respect to the specification of semantics. The pragmatic quality of SBQL is achieved by orthogonality of introduced data/object constructors, orthogonality of all the language constructs, object relativism, orthogonal persistence, typing safety, introducing all the classical and some new programming abstractions (procedures, functions, modules, types, classes, methods, views, etc.) and following commonly accepted programming languages' and software engineering principles.

SBQL queries can be embedded within statements that can change the database or program state. Typical imperative constructs are creating a new object, deleting an object, assigning new value to an object (updating) and inserting an object into another object. Typical control and loop statements such as if…then…else…, while loops, for and for each iterators, and others are also available. Some peculiarities are implied by queries that may return collections; thus there are possibilities to generalize imperative constructs according to this new feature.

SBQL in ODRA project introduces also procedures, functions and methods. All procedural abstractions of SBQL can be invoked from any procedural abstractions with no limitations and can be recursive. SBQL programming abstractions deal with parameters being any queries; thus corresponding parameter passing methods are generalized to take collections into account. The strict-call-by-value method has been implemented, which makes it possible to achieve the effects of call-by-value, call-by-reference, and more.

SBQL is a strongly typed language. Each database and program entity has to be associated with a type. However, types do not constraint semi-structured nature of the data. In particular, types allow for optional elements (similar to null values known from relational systems, but with different semantics) and collections with arbitrary cardinality constraints. Strong typing of SBQL is a prerequisite for developing powerful query optimization methods based on query rewriting and on indices.

For ODRA a generic gateway to Java libraries has been implemented. This facility allows one to use calls to Java programs within SBQL programs. The facility is especially useful to extend SBQL with GUI, with string operators, with J2EE capabilities, etc.

From the point of view of VIDE project, ODRA has been chosen as one of the exemplary target platforms to be supported by model compilers. In this role it is intended to serve for investigating the code generation issues at a purely OO database and programming language platform.

6 VIDE to Java

6.1 Approach

In the following, we map VIDE to Java. The mapping description is divided into four subsections: mapping structural aspects, mapping the behavioural parts, mapping activity diagram constructs, and finally mapping OCL expressions.

6.2 Mapping Structural Parts

VIDE data structures are given by UML class diagrams. This section is structured into three subsections, each dealing with one specification of UML *Type* metaclass: *Class*, *DataType*, and *Association*. Mapping these elements is the major task regarding the static part of VIDE, providing the environment the mapped behavioural part will be embedded in. Moreover, types are regularly used in VIDE modelling; in the context of properties within a classifier, operation signatures (including parameters, return values and exceptions) and variables in the behaviour modelling. Other aspects of VIDE structures are addressed where applicable, e.g. the package concept in the class subsection.

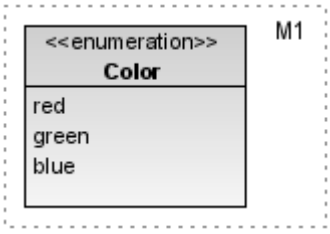
6.2.1 Data types

Instances of *DataType* are identified only by their value; typical use is to represent primitive types (e.g. *Integer*, *Boolean*) and variants of multi-valued types (e.g. *Sequence*, *Set*). In VIDE, the required simple types are adopted from the OCL extension of UML. Other types used in the VIDE PIM language (e.g. Date) are defined in a library of types that may be imported to any VIDE model.

6.2.1.1 Enumeration

Enumeration is a kind of data type that defines a finite set of literals. Enumerations are mapped to Java enum types. The enum declaration defines a class implicitly extending *java.lang.Enum*. The enumeration literals are translated into a fixed set of constant fields.

Example. Figure 25 show the mapping of an enumeration.




```
Color public enum {
    RED, GREEN, BLUE
}
```

Figure 25 : Enumeration mapping example

6.2.1.2 TupleType

TupleType is a metaclass adopted from OCL. Within instances of *TupleType*, several values of different types can be combined. These tuples are described by their composition parts (attributes); each part can be uniquely identified by its name. As tuples are specified to be immutable, instances of *TupleType* and their values are created at the same time. Tuples may be compared based on their name and value of their attributes.
Directly mapping the metaclass *TupleType* requires some kind of lightweight data structure in the target language, with similar value handling as e.g. in Java primitive types. However there is no equivalent concept in Java. A working solution is to implement tuples as instances of a class named *Tuple*. This class controls access to an instance of *java.util.Map<String, Object>* that holds the names and values of the tuple attributes. The immutability of the map entries can be ensured by making the mutator method of the Map protected.

Example. Listing 6.1 shows an OCL tuple. Applying the proposed mapping results in the Java code fragment shown in Listing 6.2.

```
Tuple {name: String = 'John', age: Integer = 10}
```

Listing 6.1: Tuple mapping example (OCL statement input)

```
Tuple t1 = new Tuple(){{set("name", "John"); set("age",
10);}};
```

Listing 6.2: Tuple mapping example (output)

6.2.1.3 Primitive types

VIDE models use the primitive types provided by OCL. OCL defines four basic types inheriting from *PrimitiveType*: *Integer*, *Real*, *Boolean*, and *String*. Transferring these types to equivalent Java constructs is straightforward; they can be mapped directly to Java primitive types, as shown in Table 2. However, there are conceptual discrepancies as in OCL everything is considered an object. This becomes apparent in the context of collections – OCL knows collections of basic types, whereas Java allows only object reference collections. The autoboxing feature in Java 5 hides this problem when accessing and manipulating primitive type collections. Nevertheless, the Java Wrapper type has to be used in the collection declaration statement.

OCL basic type	Java type	Java Wrapper type	Java default value
Boolean	Boolean	java.lang.Boolean	false
Integer	Int	java.lang.Integer	0
Real	Double	java.lang.Double	0.0d
String	String		null

Table 2: Mapping of OCL basic types

6.2.1.4 Collection types

Collection types are data types that can contain multiple elements of a specific type. Similar to the basic types in the last section, collection types are obtained from the OCL standard library. The library contains four implementations of the abstract *CollectionType* class. *Bag* instances may contain duplicates and have no ordering; a *Sequence* is an ordered bag; a *Set* is a bag without duplicates; and finally an *OrderedSet* has both unique and ordered elements.

There is an additional way to model collections. All subclasses of the metaclass *MultipleElement* (*Property*, *Parameter*, and *Variable*) own the properties *upper* and *lower* that specify the number of contained elements. Additionally, the mentioned metaclasses have the properties *isUnique* and *isOrdered*, resulting in the four variants that are similar to the listed subtypes of OCL *CollectionType*.

In Java, the *java.util.Collection* classes can be used as equivalent counterparts to the variants of OCL *CollectionType*, shown in Table 3, as well as for the mapping of UML *MultipleElement*s, listed in Table 4. The tables provide suitable abstract declaration types and additionally instantiation types that implement the collection interfaces used for declaration. The instantiation types can be obtained from the predefined Java collection types with one exception, as Java does not contain an adequate concept for ordered and unique multi-valued types. The *List* interface can be used for declaration, as it provides the necessary accessor and mutator functionality. But the predefined implementations of *List*, e.g. *java.util.ArrayList*, do not ensure element uniqueness. *java.util.SortedSet* might be used in this context. *SortedSet* implements the *Set* interface and additionally introduces element order. However, this order is strictly ascending as far as a comparator is concerned. Consequently, *SortedSet* provides functionality to access the first and the last element of the ordering, but no direct access to other positions. This mapping therefore introduces an own implementation *vide.UniqueList* of the *java.util.List* interface that ensures uniqueness when an element is added.

OCL	**Java**	
collection type	**declaration type**	**instantiation type**
Bag(T)	java.util.Collection(? extends T)	java.util.ArrayList(? extends T)
Sequence(T)	java.util.List(? extends T)	java.util.ArrayList(? extends T)
Set(T)	java.util.Set(? extends T)	java.util.HashSet(? extends T)
OrderedSet(T)	java.util.List(? extends T)	vide.UniqueList(? extends T)

Table 3: Mapping of OCL collection types

UML			Java	
type	**ordered**	**unique**	**declaration type**	**instantiation type**
T	false	false	java.util.Collection(? extends T)	java.util.ArrayList(? exrends T)
T	true	false	java.util.List(? extends T)	java.util.ArrayList(? extends T)
T	false	true	java.util.Set(? extends T)	java.util.HashSet(? extends T)
T	true	true	java.util.List(? extends T)	vide.UniqueList(? extends T)

Table 4: Mapping of UML multiple elements

6.2.2 Classes and packages

In UML, packages are used to group elements and provide a shared namespace. Packages are the primary mean of UML for the decomposition of complex models. Among other subtypes of *PackageableElement*, packages can recursively own packages resulting in a tree structure. Moreover, packages can own instances of the UML metaclass *Class*. Classes are the blueprint for objects that share the same features, constraints, and semantics. Therefore, classes have a name and a set of properties and operations. With regard to the organisation of classes, an important concept is the generalisation relationship between classes. It allows reusing the characteristics of classes, as instances of class *A* can be viewed as instances of class *B* as well, if *B* is a generalisation of *A*. UML does not restrict classes to have only a single generalisation.

Mapping UML packages and classes to Java is straightforward, as Java also uses classes, organised in packages. The names of the resulting Java packages and classes are taken from the modelled names. Though metaclass Interface is discarded in the VIDE modelling language, it is useful to translate abstract UML classes to Java interfaces, if they only contain constants and operations that do not contain behaviour. Other UML classes are mapped to Java implementation classes and, additionally, to interfaces named like the classes, but with a preceding “I”. These interfaces are used for variable declaration.

The UML generalisation semantics can be transferred to the Java concept of inheritance. However, Java supports multiple inheritance only with regard to interfaces, and not with regard to implementation classes. VIDE allows multiple inheritance, but treats any inheritance conflict as an error. This simplifies the mapping, as e.g. naming conflicts or the diamond problem are not relevant. Based on this, the mapping strategy described in the following is sufficient.

All generalisations of the class *A* are listed in the association *superClass*. Class *A* may extend one class (called *extension*) and may inherit from a list of interfaces (called *interfaceList*). In the following, the elements of *superClass* are mapped to the extension and interfaces:

1. All elements of *superClass* that are mapped (only) to an interface are added to the *interfaceList*.
2. From the remaining elements of *superClass* the first one is used as *extension*.
3. For the remaining classes the corresponding interfaces (named as the modelled class with a preceding “I”) are added to the *interfaceList*. Additionally, the properties and operations of these superclasses are added to the class body of *A*, if *A* does not contain equally typed and named properties, respectively operations with the same signature.

Example. Figure 26 shows an UML class hierarchy with class *A1* and *A2* being generalisations of class *B*. Both *A1* and *A2* are non-abstract. With the mapping provided above, this class hierarchy is transferred to the hierarchy in Figure 27. Listing 3 shows the Java output for class *B*.

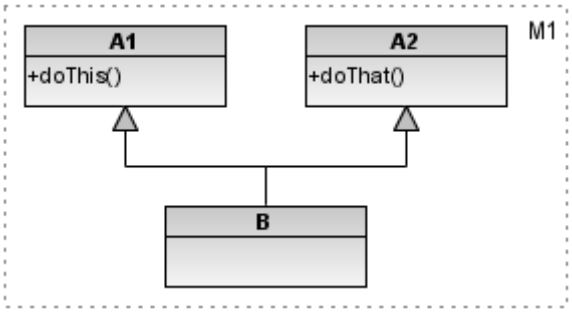


Figure 26 : Class inheritance mapping example (input)

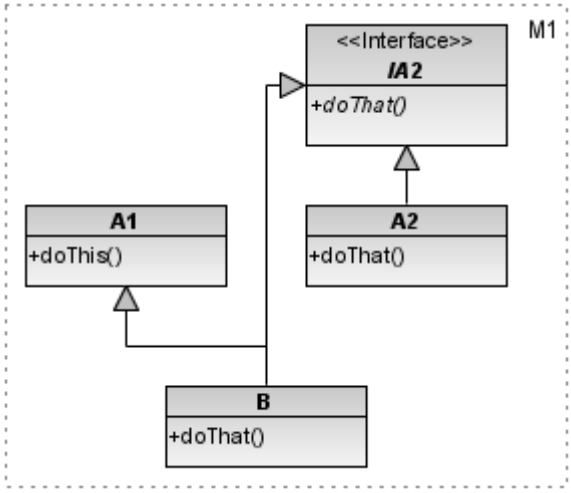


Figure 27 : Class inheritance mapping example (transformed)

```
public class B extends A1 implements A2 {
    public void doThat(){
        ...
    }
}
```

Listing 3: Class inheritance (output of class B)

6.2.3 Association

An association is a relationship that can occur between typed instances. It has at least two *ends* represented by properties, each of which is connected to the type of the *end*. Instances of association are called links. Associations are one of the major abstraction concepts in object-oriented modelling. However, there is no equivalent concept in Java language. Therefore, mapping the association structures and semantics to Java must be transferred to adequate classes, attributes and methods. VIDE modelling language only makes use of binary associations and excludes qualified associations and association classes. The two end properties are therefore mapped to attributes with accessor and mutator functionality in the involved classes, if the opposite *end* is navigable. In the bi-directional case, the mutator methods must ensure the synchrony of the opposite link end.

6.2.4 Property

In UML, a property may occur as an attribute of a class as well as an *end* of an association. Properties have a ***name***, which is directly adopted by the Java attribute, and a ***type***. The UML type can be a *DataType* (cf. Section 6.1.1) or a class contained in the model. If the *upper* characteristic is greater than one, the Java type additionally depends on the **multiplicity** of the property. The resulting Java type is then a generic collection, as defined in Table 4, with the mapped property type as element type.

Mapping the four **visibility** modifiers (public, protected, private, package) of UML Properties to Java is straightforward, as Java has the same access control levels for attributes with almost the same generated semantics. Nevertheless, properties are always mapped to private attributes to support data encapsulation. If the visibility of the property is not private, accessor and mutator methods are produced as described in this section. The access modifier of these methods is mapped from the visibility of the property. Additionally, there is a restriction concerning the visibility used in bi-directional associations – mutator methods of involved *ends* are public as they must be accessible from the opposite side for synchronisation.

The property may be declared ***readOnly*** and ***isStatic,*** which is mapped to the *final* and *static* modifiers in Java.

The following tables define a common interface of accessor and mutator method signatures for properties. As several actions deal with accessing and manipulating properties and associations this interface abstracts from the different variants of properties.

Short name	Method signature and description
ObjectGet	`+getProp() : T`
	Returns the value of *prop*.
ObjectSet	`+setProp(T newValue) : void`
	Sets the attribute to *newValue*. If the mapped property is *memberEnd* of a bi-directional association, both old and new values have to be synchronised. In case of a multi-valued property, this update has to be performed for all elements of the old and new collection.

Table 5: Basic field accessor and mutator methods

Table 5 lists the accessors and mutators that are generated for all properties. The method signatures assume that the property is identified as *prop* and its type is mapped to the Java type *T*.

If the property *prop* is multi-valued, additional methods should be produced as described in Table 6. It is assumed that the UML type of the elements of *prop* can be mapped to Java type *E*.

Short name	Method signature and description
ObjectAdd	`+addToProp(E value) : boolean`
	Tries to add *value* to the attribute. The return value indicates, whether or not the addition was successful. In case of a bi-directional association, both old and new values have to be synchronised.
ObjectRemove	`+removeFromProp(E value) : boolean`
	Tries to remove *value* from the attribute. Returns whether or not

	the removal was successful. In case of a bi-directional association, the involved instances must be updated.
RemoveAll	`+removeAllFromProp() : void`
	Removes all elements from the attribute and updates the links in case of a bi-directional association.
HasIn	`+hasInProp(E value) : boolean`
	Checks, if value is an element of the attribute.
IteratorOf	`+iteratorOfProp() : java.util.Iterator<T>`
	Returns an *Iterator* over all elements of the attribute.
SizeOf	`+sizeOfProp() : int`
	Returns the number of elements of the attribute.

Table 6: Additional accessor and mutator methods for multi-valued fields

Furthermore, if the multi-valued property *prop* is ordered, the methods listed in Table 7 should be generated additionally.

Short name	Method signature and description
PositionGet	`+getProp(int position) : E`
	Returns the element at *position* from the *prop* collection. If this element is not an instance of *DataType*, the result depends on whether or not the element is declared as destroyed. If *isDestroyed()* of the element returns true, the element is removed from the list and PositionGet returns null. Otherwise, a reference to the element is returned.
PositionAdd	`+addAtIndexToProp(int position, T newValue) : void`
	Adds *newValue* to the multi-valued attribute at *position* and updates the links in case of a bi-directional association.
PositionRemove	`+removeAtIndexFromProp(int position) : void`
	Removes element at position i from the attribute. In case of a bi-directional association, the removed instance has to be updated, too.

Table 7: Additional mutator methods for multi-valued, ordered fields

The methods listed in Table 8 are not part of the common accessor and mutator interface. They are only generated in the context of ordered association ends to support the link actions *CreateLinkAction* and *DestroyLinkAction*.

Short name	Method signature and description
DoublePositionAdd	`+addAtIndexToProp(int pos1, T newValue, int pos2) : void`
	Adds *newValue* to the own association end attribute at position *pos1* and causes the *newValue* instance to add the calling object to its association end attribute at *pos2*.
DoublePositionRemove	`+removeAtIndexFromProp(int pos1, int pos2) : void`
	Removes element at position *pos1* from the own association end attribute and causes this element to remove the target object from its own association end attribute at position *pos2*.

Table 8: Additional mutator methods for multi-valued, ordered association ends

6.2.5 Operation

An operation is a behavioural feature of a classifier that specifies the name, type, parameters, and constraints for invoking an associated behaviour. UML operations are mapped to method declarations in Java. The method body is derived from the associated activity. Mapping the **visibility** of the operation to an equivalent Java access modifier is straightforward, as shown in the last section. Though VIDE does not adopt the *isAbstract* characteristic of UML, operations are mapped to **abstract** method calls, if the class is modelled to be abstract and no activity is linked to the operation. The method **name** is derived from the *name* property of the UML operation; however it has to be checked if the method has the same signature as the accessor and mutator methods defined in the last section. Conflicts should be resolved by adapting the method's name, e.g. by appending the String "Modelled". If the name of the operation is equal to the name of the class then mapping this operation results in a Java class constructor.

The method return type as well as the method arguments depend on the **parameters** contained in the operation. Parameters have four kinds of direction: *in*, *inout*, *out*, and *return*. A single parameter may be distinguished as a return parameter. The type and multiplicity of this parameter are mapped to the Java return type of the method. The parameters with direction kind *in* are translated to the method arguments. However, there is no direct conceptual counterpart for the remaining two parameter directions *inout* and *out*, as Java method may not have multiple return types. To overcome this limitation, this mapping introduces additional attributes in the class of the operation. In case of direction *inout*, these attributes are initialised with the argument value that is to be passed into the method. After method execution, the manipulated attribute is readout. In case of direction *out*, only the latter step is performed.

Example. Figure 28 shows the a class *Taxi* with an operation *drive()* and a parameter of each direction kind. Listing 4 shows the resulting attributes and methods in the Java class *Taxi*, and Listing 5 provides exemplary code that is used to call the method *drive()*.

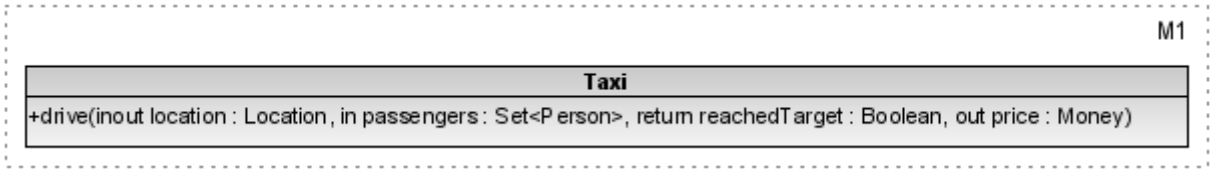


Figure 28 : Operation mapping example (input)

```
private Location location;
public int getLocation() { return location; }
public void setLocation(double newValue) { location =
newValue; }
private double price;
public Money getPrice() { return price; }
public boolean drive(Set<Person> passengers){…}
```

Listing 4: Operation mapping example (output – attributes and methods)

```
aTaxi.setLocation(someLocation);
Boolean arrived = aTaxi.drive(setOfPersons);
someLocation = aTaxi.getLocation();
Money price = aCar.getPrice();
```

Listing 5: Operation mapping example (output – method calls)

6.3 Mapping Actions to Java

This section defines how the UML actions that are included in the VIDE metamodel [Ref. D2.1] can be mapped to Java statements. The definitions of the different actions were taken from D2.1 but they were extended to also show inherited attributes and associations as these are relevant for mapping the actions to Java statements.

6.3.1 Sample input model for Actions

To exemplarily demonstrate the code fragments resulting from the specified actions mapping, the following sample input model is introduced:

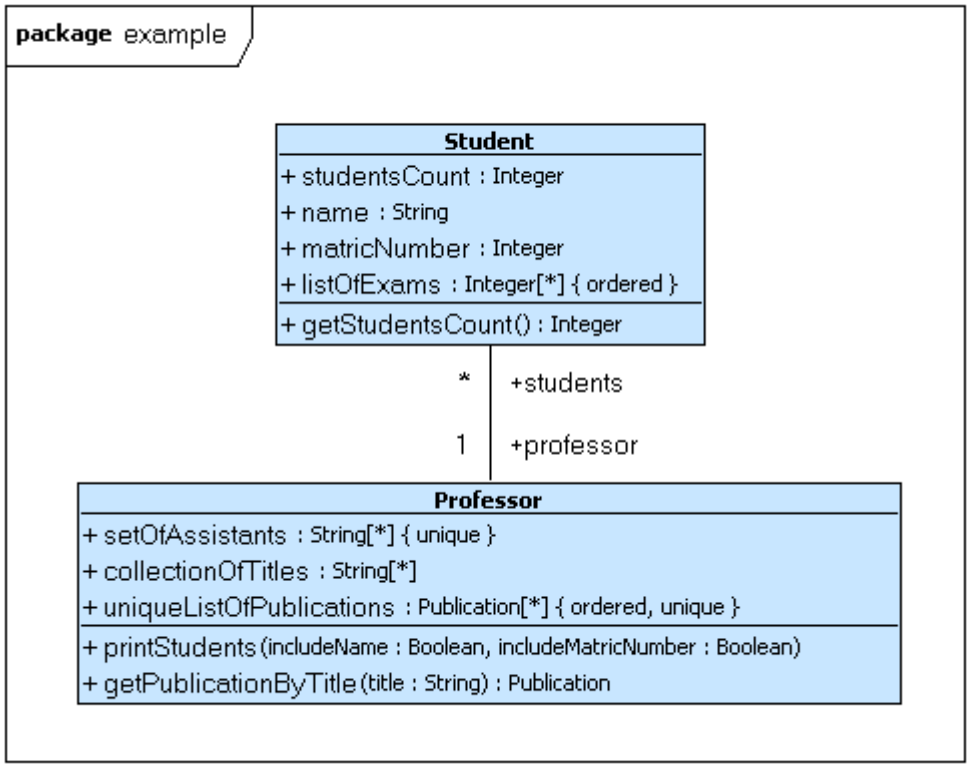


Figure 29 : Example model

The multiplicity types of the structural features are chosen to cover all potential combinations of the properties isUnique and isOrdered. The setting of these structural features can be determined by the prefix of their name:

Prefix	isOrdered	isUnique
collectionOf	False	false
listOf	True	false
setOf	False	true
uniqueListOf	True	true

Table 9: Multiplicity table of the example

If for demonstration purposes multiple-valued local variables or parameters are introduced, they are named accordingly.

6.3.2 General Concepts

6.3.2.1 Action

As Action is an abstract class, no direct mapping to Java code is provided. The mapping of the subclasses inheriting from the metaclass Action is described in the following subsections of this chapter.

6.3.2.2 Object Flow

In an activity, objects can be passed between the contained actions using output pins, object flows and input pins. The mapping rules of object flow are relevant for many Actions. Therefore, we define in the following a mapping strategy for pins and object flows.

Mapping of InputPins, ObjectFlows and OutputPins

To reflect the flow of objects in the generated Java code, a table is used to keep track of the objects that are made accessible via object flows while traversing an action graph. This table contains a map of each object flow contained in the activity to a reference to the object flow's object.

Initialization of the object flow table:
The table is created when the UML Actions to Java generator enters an activity. All object flows found in the activity are added to the table as "empty" flows (respective object references are set to null). Additionally, for all ActivityParameterNodes contained in the activity the outgoing object flows are updated in the list to have a reference to the object specified in the ActivityParameterNode.

Reading the object flow table (InputPins):
If the generator traverses an Action with InputPins the referenced object can be accessed using the object flow table. The incoming property of the InputPin specifies the object flow entry in the table. The corresponding object reference is used as input object for the Action.
(ValuePins are accessed in another way; see "Mapping of ValuePins" section)

Updating the object flow table (OutputPins):
If there are OutputPins contained in the traversed Action, the object flows specified in the outgoing property of the OutputPin are updated with a reference to the object that is defined as output object in the Action specification.

Mapping of ValuePins

A ValuePin is treated differently from other InputPins, as this Pin is used to integrate an expression in the code. This expression is of type ValueSpecification and appears in two different (subclass-)variants in VIDE: OpaqueExpression and ExpressionInOCL.

OpaqueExpression:
In case of an OpaqueExpression the body property is read, which is a list of Strings. The first list element is used as value of the ValuePin in the code generation process. Additional elements of the body property are discarded.

ExpressionInOCL:
An ExpressionInOCL is the root element for an expression specified with OCL model elements. This subtree has to be traversed to generate the Java String corresponding to the expression. This String is integrated in the code generation process as value of the ValuePin.

6.3.3 Invocation Actions

6.3.3.1 CallOperationAction

The generator takes the name *op* of the Operation associated to the *CallOperationAction* and uses it in an operation call statement in Java.

If the Operation is static then the name of the method call statement should include the name of the class where the method is defined. The information on whether an operation is static or not is available using the attribute *isStatic*. The name of the class where a static method is defined can be accessed using the association between the metaclasses *Operation* and *Class*.

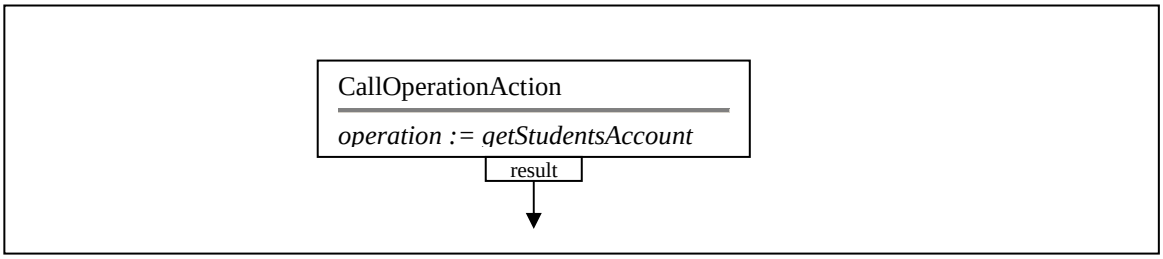
If the Operation is not static, the name of the target object instance has to be included in the generated Java method call statement. The name can be get from the target input pin of the action (cf. Chapter 1.1.2).

For each argument input pin contained in the argument list of the action the name of the referenced object has to be included as parameter in the generated Java method call statement.

If the Operation has a return type (determined by the existence of a contained parameter with direction set to *return*) then a new temporary variable is declared (cf. chapter 1.1.4) and the method call statement in Java is generated as the right side of an assignment statement to that new variable. Additionally, a reference to the result object is stored in the outputPin (cf. chapter 1.1.2).

Examples (based on the sample model introduced in Appendix A):

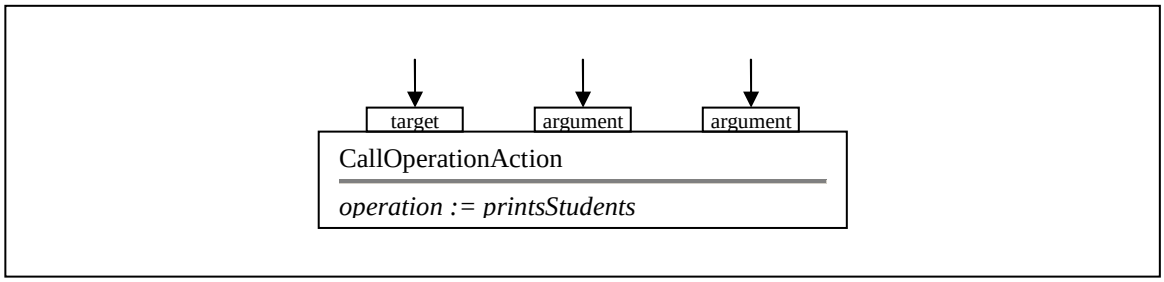
1) A CallOperationAction of the operation *getStudentsCount*, containing a result outputpin:



The static Operation *getStudentsCount* is contained in the Class *Student* and has a parameter of type *integer* and direction *return*. Therefore, the CallOperationAction results in the following Java code fragment:

```
int var__1 = Student.getStudentsCount();
```

2) A CallOperationAction of the operation *printStudents*, containing a target InputPin and two argument InputPins.



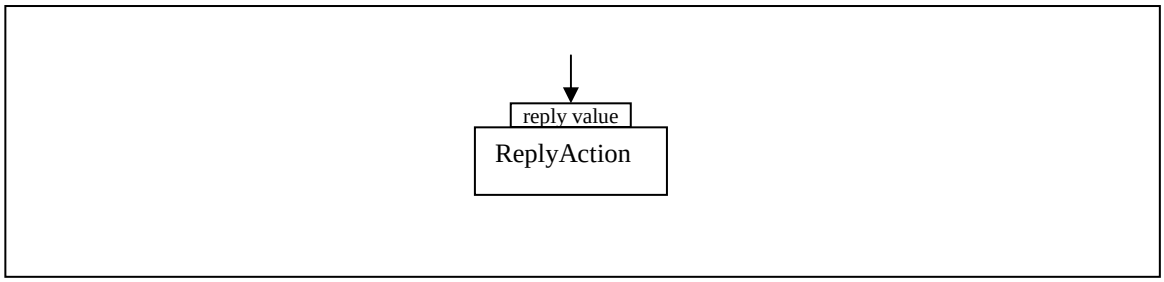
The operation *printStudents* is not static and has no Parameter of direction return. Assuming, that via target InputPin an instance of class *Professor* with identifier *professorA* is accessible and both argument InputPins can be resolved to the boolean value *true*, the following code fragment will be generated:

```
professorA.printStudents(true, true);
```

6.3.3.2 ReplyAction

A return statement in Java (“return”) is generated from this action and the statement generated for the first replyValue statement is appended. As Java does not support multiple return values in a return statement, only the first reply value is taken into account. The others are discarded.

Example (based on the sample model introduced in Appendix A):



ReplyAction containing a reply value InputPin:

Assuming, that the reply value InputPin can be resolved to an object with the identifier *value*, the following code is generated:

```
return value;
```

6.3.4 Object Creation Actions

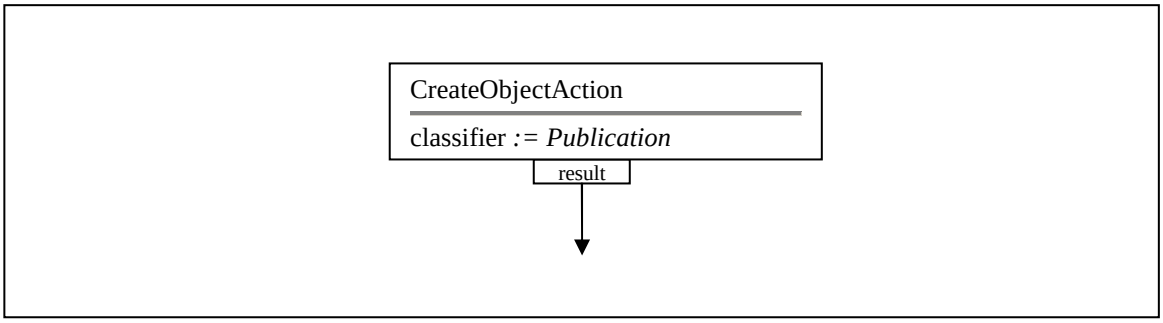
6.3.4.1 CreateObjectAction

This action is mapped to a constructor call statement in Java. The name of class (which is also the name of the constructor method in Java) is accessible through the association to the metaclass Classifier. According to the definition of this action, no parameters can be passed.

Similarly to method calls with return values a new variable that has the same type as the constructor class (i.e., the classifier referenced in the action) is declared and an assignment statement is generated so that the constructor call statement is assigned to that new variable. A unique identifier that is not already used in the models is used for that new variable (cf. Section 1.1.4).

Example :

A CreateObjectAction with its classifier set to the Class Publication:



Assuming, that the name property of the result OutputPin is set to *result*, the following Java code is generated:

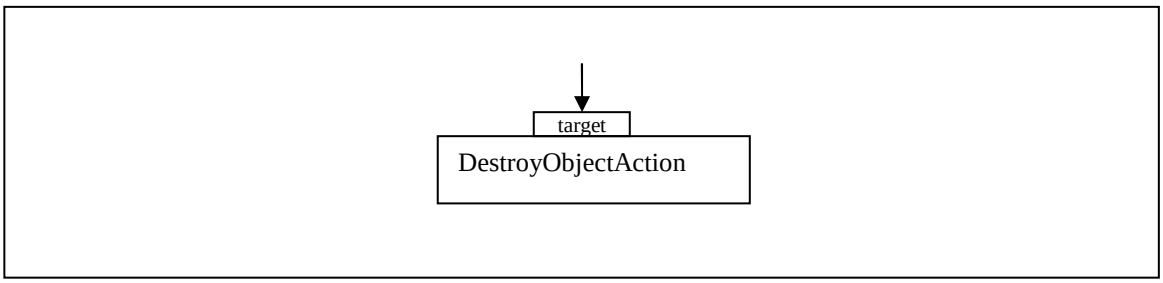
```
Publication result = new Publication();
```

6.3.4.2 DestroyObjectAction

Java does not provide destructors. The garbage collector automatically determines, what data objects are no longer accessed and reclaims the resources used by these objects. Therefore, a DestroyObjectAction is mapped to an assignment of null to the object reference specified with the target InputPin.

Example :

A DestroyObjectAction containing a target InputPin:



If the target InputPin can be resolved to an object reference named *var*, the following code will be generated:

```
var = null;
```

6.3.5 StructuralFeature Actions

Figure 3 shows the structural feature actions in UML2. In the following, we will describe the mapping of the actions to Java.

The structural feature specified in these actions can be a property of the object or an association end. With the interface of accessor and mutator methods defined in section 1.1.3, there is no need to distinguish between these two kinds of structural features. If one of these methods is referenced in the following subsections, the short name defined in the tables of sections 1.1.3 will be used (e.g. “Getter”, “ObjectRemove”).

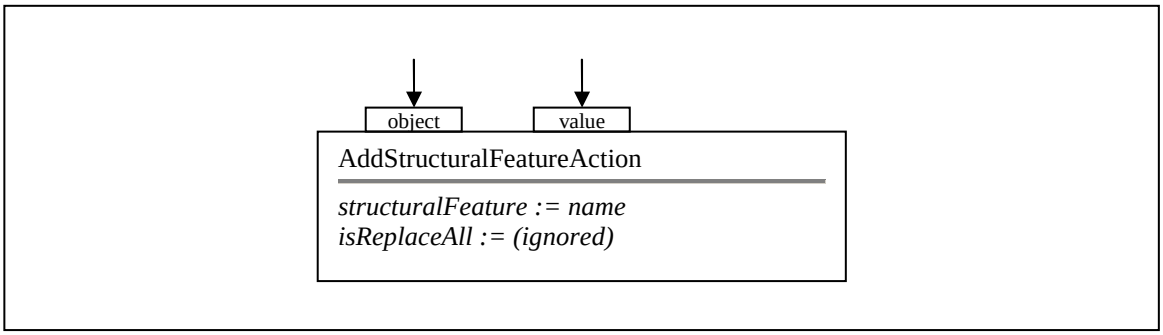
6.3.5.1 AddStructuralFeatureValueAction

This action is mapped to an assignment statement in Java. To generate the left side of the assignment the object input pin is used for generating the object reference string and the structural feature name is used for generating the attribute name in Java. The right-side of the assignment is generated using the value input pin. However, the generated code depends on the multiplicity of the structural feature (determined from its “Upper” value). If the feature is multi-valued, several cases have to be differentiated based on the structural feature’s attribute “isOrdered” and the action features “isReplaceAll” and “insertAt”:

- Non-multiple Structural Feature: a normal assignment via Setter method is generated. (example 1)
- Multiple Structural Feature:
    - replaceAll = true: generates a replacement via Setter method. (example 2)
    - replaceAll = false:
        - not ordered: generates an insertion using the ObjectAdd method. (example 3)
        - ordered:
            - insertAt = null: generates an insertion at the end of the list using the ObjectAdd method. (generated output is equivalent to example 3)
            - insertAt = *pos*: generates an insertion at position *pos* using the PositionAdd method. (example 4)

Example :

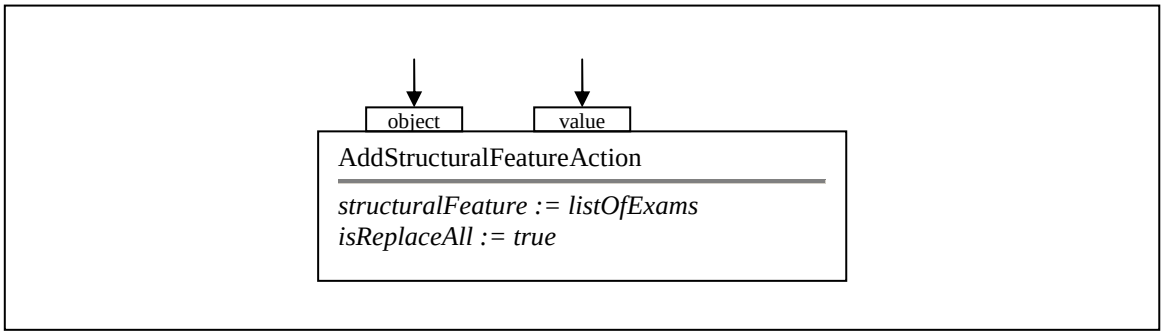
1) An AddSructuralFeatureAction with a non-multiple structural feature:



Assuming that the object InputPin can be resolved to a Student *studentXY* and the value InputPin can be resolved to the String *"Anton"*, the generated code is then:

```
studentXY.setName("Anton");
```

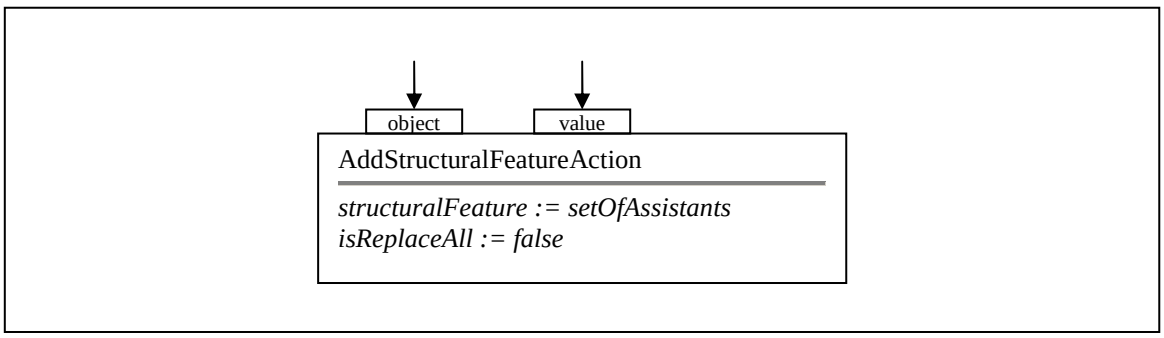
2) An AddStructuralFeatureAction with a multiple-valued structural feature and the property isReplaceAll set to true:



Assuming that the object InputPin can be resolved to a Student *studentXY* and the value InputPin resolves to a list of ExamIDs *someExamList*, the generated code is:

```
studentXY.setListOfExams(someExamList);
```

3) An AddStructuralFeatureAction with a multiple-valued structural feature and the property isReplaceAll set to false. Additionally, the Action contains an insertAt InputPin:



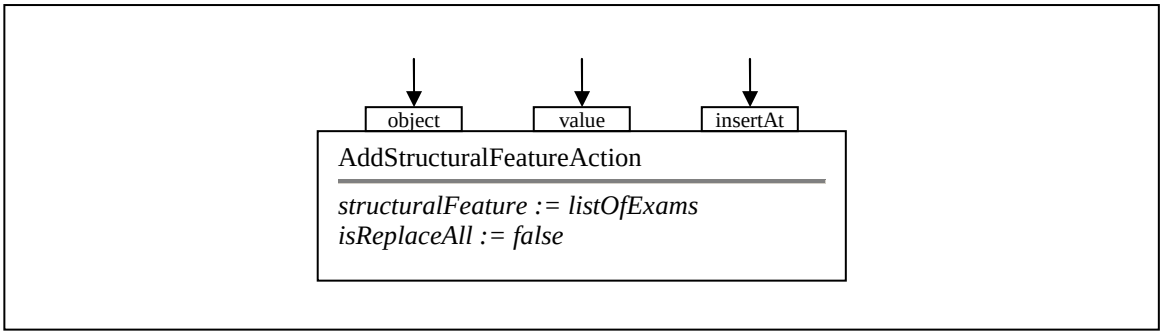
Assuming that:

- the object InputPin can be resolved to a Professor *professorXY*
- the value InputPin can be resolved to a String with identifier *aString*

The following code will be generated:

```
professorXY.addToSetOfAssistants(aString);
```

4) An AddStructuralFeatureAction with a multiple-valued structural feature and the property isReplaceAll set to false. Additionally, the Action contains an insertAt InputPin:



Assuming that:
- the object InputPin resolves to a Student *studentXY*
- the value InputPin resolves to an integer *examID*
- the insertAt InputPin resolves to an integer *indexPos*

The following code is generated:

```
studentXY.addAtIndexToListOfExams(indexPos, examID);
```

6.3.5.2 ClearStructuralFeatureValueAction

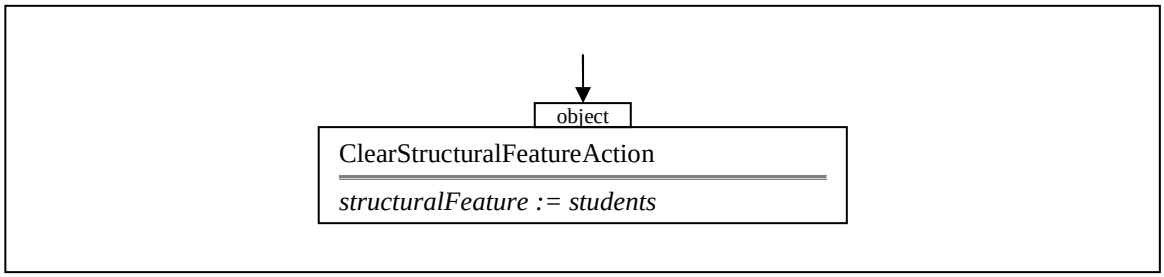
The mapping of this action to Java depends on the multiplicity of the structural feature. If it is multi-valued, the method *removeAll* is called on the structural feature (example 1).

If not multiple valued, a Setter method call is generated, In the case of a non-primitive type the attribute value is set to null (example 2). Otherwise, the Java default value (given in the table below) of the respective primitive type is used:

Primitive Type	Default Value
byte, short, int	0
long	0L
float	0.0f
double	0.0d
char	‘\u0000’
boolean	false

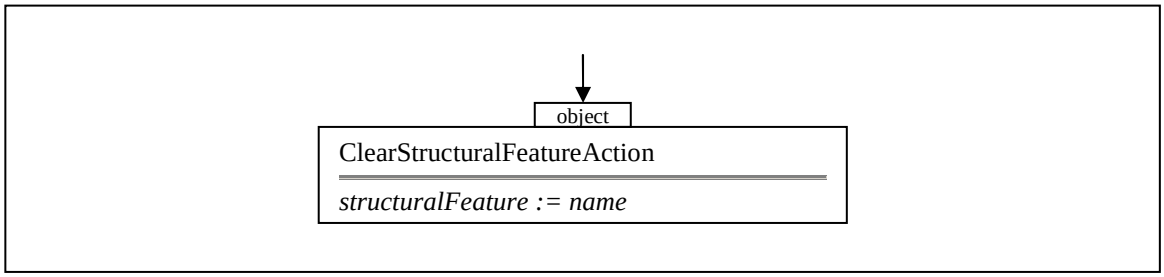
Example :

1) A ClearStructuralFeatureAction with a multiple-valued structural feature:



If the object can be resolved to a Professor instance with the identifier *professorXY*, the following code fragment is generated:

```
professorXY.removeAllFromStudents();
```



2) A ClearStructuralFeatureAction with a non-multiple-valued structural feature:

If the object can be resolved to a Student instance with the identifier *studentXY*, the following code fragment is generated:

```
studentXY.setName(null);
```

6.3.5.3 RemoveStructuralFeatureValueAction

If the structural feature is not multi-valued (upper == 1), this action is treated similarly to ClearStructuralFeatureAction (see 1.4.2).

If the structural feature is multiple, four cases can occur depending on the values of the properties isUnique and isOrdered:

1) unique and ordered (Unique List):

isRemoveDuplicates is ignored because the List is already unique.

a) removeAt = i: a PositionRemove method call is generated. An object possibly specified as value of the value input pin is ignored (example 1).

b) removeAt = null: an ObjectRemove method call is generated (example 2).

2) unique and unordered (Set):

isRemoveDuplicates is ignored because the Set is already unique.

removeAt is ignored because the Set is unordered.

An ObjectRemove method call is generated (analog to example 2)

3) not unique and ordered (List):

a) removeAt = i

I. isRemoveDuplicates = true: First, a new, temporary variable (cf. chapter 1.1.4) of the same type as the structural feature elements is declared and initialized with the structural feature element at position removeAt. Afterwards, a PositionRemove-method call is generated, as a test statement in an empty while loop. The new variable serves as parameter of the remove method. An object possibly specified as value of the value input pin is ignored. (example 3)

II. isRemoveDuplicates = false: A PositionRemove method call is generated (analog to example 1). An object possibly specified as value of the object input pin is ignored.

b) removeAt = null

I. isRemoveDuplicates = true: An ObjectRemove-method is called as a test statement in an empty while loop. The object specified with the value InputPin is the parameter of this method. (example 4)

II. isRemoveDuplicates = false: an ObjectRemove method call is generated (analog to example 2)

4) not unique and unordered (Collection):

removeAt is ignored because the Collection is unordered.

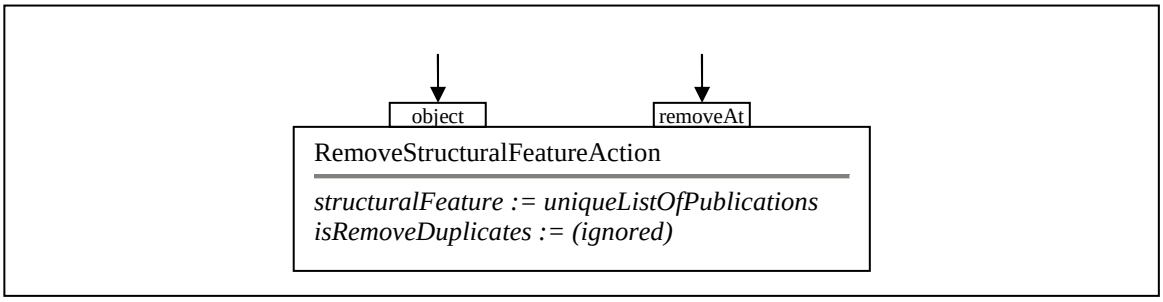
Comparable to case 3b.

a) isRemoveDuplicates = true: An ObjectRemove-method is called as a test statement in an empty while loop. The object specified with the value InputPin is the parameter of this method. (analog to example 4).

b) isRemoveDuplicates = false: an ObjectRemove method call is generated (analog to example 2).

Example :

1) A RemoveStructuralFeatureAction with a unique-list kind of structural feature and a contained removeAt InputPin:

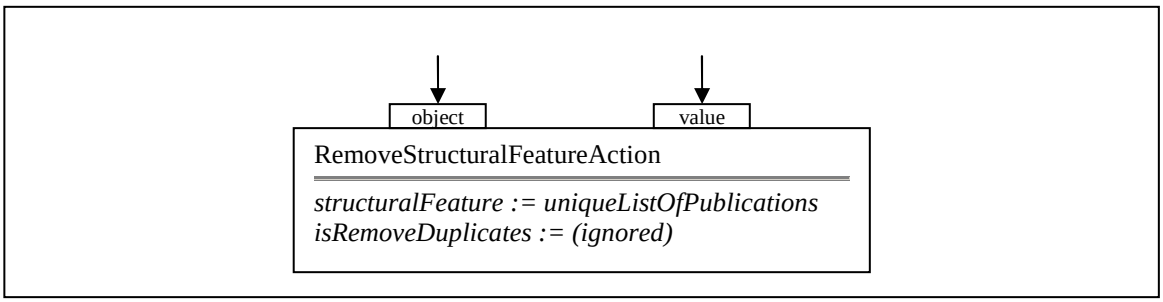


Assumptions:

- the object InputPin can be resolved to a Professor *professorXY*
- the removeAt InputPin resolves to an Integer *indexPos*

The following code is generated:

```
professorXY.removeAtIndexFromUniquelistOfPublications(indexPos);
```



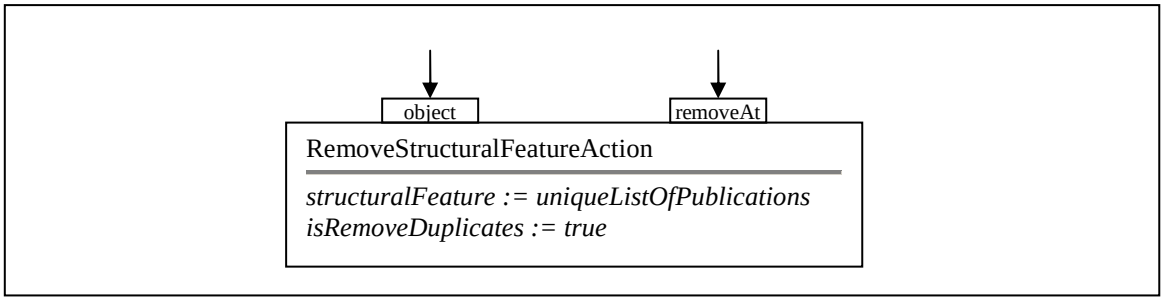
2) A RemoveStructuralFeatureAction with a unique-list kind of structural feature, without removeAt InputPin:

Assumptions:
- the object InputPin can be resolved to a Professor *professorXY*
- the value InputPin can be resolved to a Publication *aPublication*.

The following code is generated:

```
professorXY.removeFromUniquelistOfPublications(aPublication);
```

3) A RemoveStructuralFeatureAction with a non-unique kind of structural feature and a removeAt InputPin, isRemoveDuplicates set to true:

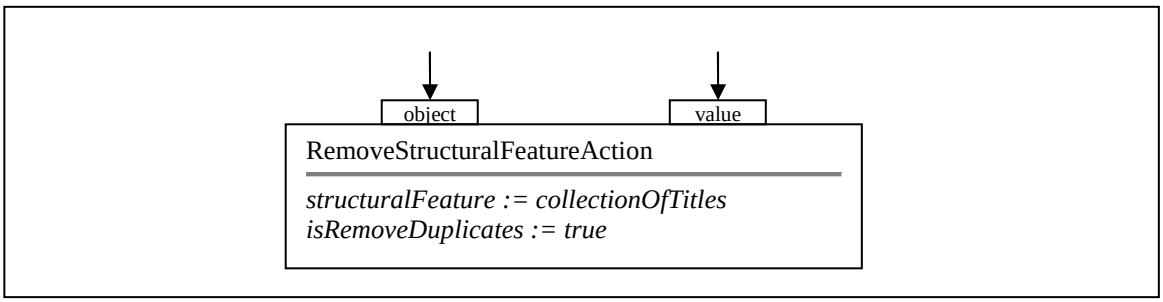


Assumptions:
- the object InputPin can be resolved to a Professor *professorXY*
- the removeAt InputPin can be resolved to *7*.

The following code is generated:

```
Publication var__1 =
professorXY.getUniqueListOfPublications().get(7);
while (professorXY.removeFromUniqueListOfPublications (var__1)){}
```

4) A RemoveStructuralFeatureAction with a non-unique kind of structural feature, isRemoveDuplicates set to true:



Assumptions:

- the object InputPin can be resolved to a Professor *professorXY*
- the value InputPin can be resolved to a String *aTitle*.

The following code is generated:

```
while (professorXY.removeFromCollectionOfTitles(aTitle)){}
```

6.3.6 Link Actions

Link actions are only used in the context of associations. As the interface of accessor and mutator methods defined in section 1.1.3 are generated for both properties and association ends, this functionality can be used in the following mapping rules (referenced by the short name defined in the tables of sections 1.1.3).

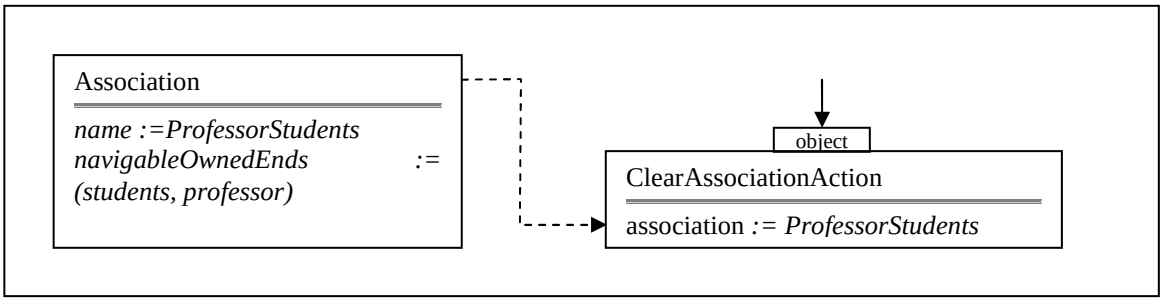
6.3.6.1 ClearAssociationAction

ClearAssociationAction is mapped to RemoveAll method calls on the object passed to the action via the object InputPin.

Associations contained in VIDE models are limited to a maximum of 2 navigable association ends, which are listed in the association's navigableOwnedEnd property. To determine the respective association end for the object input pin, the type of that object is compared with the type of both association ends found in navigableOwnedEnd. For each type match, the elements associated with the input pin object are deleted by generating a call to the respective RemoveAll method. Note, that the input pin object can be a member of both ends in case of a looping association is looping.

Example :

A ClearAssociationAction with an object InputPin:



Assumptions:

- the object InputPin can be resolved to a Professor *professorXY*

This is mapped to the following code fragment:

```
professorXY.removeAllFromStudents();
```

6.3.6.2 CreateLinkAction

The mapping of a CreateLinkAction depends on the information given with the two associated LinkEndCreationData (in the following called A and B) elements.

If the properties specified by A and B are non-multiple, the CreateLinkAction is mapped to a Setter method call with the value InputPins of A and B respectively as target and parameter (example 1).

If the properties specified by A and B are multiple-valued and isOrdered is set to false, the insertAt InputPin is ignored (the "insertAt = null" case is described in the following mapping).

Moreover, the property isReplaceAll of A and B and their optional insertAt InputPin have to be taken into account:

- isReplaceAll = false for both:
  - insertAt = null for both:
    An ObjectAdd method call is generated. The target and parameter of this method is specified by the value InputPins of A and B (example 2).
  - insertAt not null for A, insertAt = null for B
    A PositionAdd method call is generated. The target is specified by the value InputPin of B, the position parameter by the insertAt InputPin of A, the object parameter by the value InputPin of A (example 3).
  - insertAt = null for A, insertAt not null for B
    (symmetric to the last case)
  - insertAt not null for both
    A DoublePositionAdd method call is generated. The target is specified by the value InputPin of B, the first position parameter by the insertAt InputPin of A, the object parameter by the value InputPin of A, the second position parameter by the insertAt InputPin of B (example 4).
- isReplaceAll = true for A, isReplaceAll = false for B
  (insertAt of A is ignored)
  - insertAt = null for B

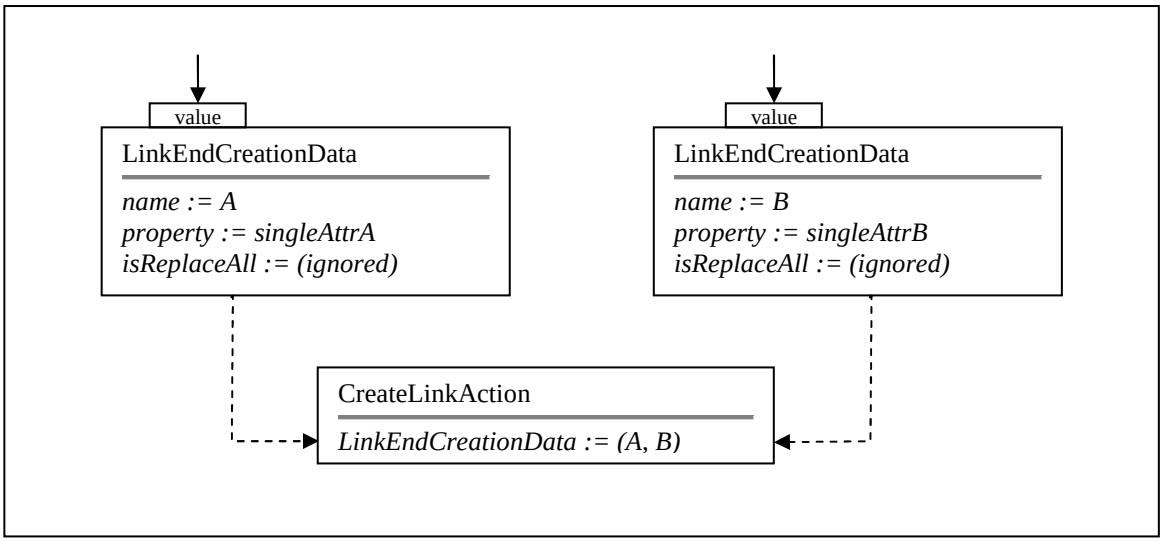
A RemoveAll method call is generated with the value InputPin of B as target. Afterwards, the AddObject method call is generated, taking the two objects specified by the value InputPins as target and parameter (example 5).

- insertAt not null for B
  A RemoveAll method call is generated with the value InputPin of B as target. Afterwards, a PositionAdd method call is generated. The target is specified by the value InputPin of A, the position parameter by the insertAt InputPin of B, the object parameter by the value InputPin of B (example 6).

- isReplaceAll = false for A, isReplaceAll = true for B
  (symmetric to the last case).
- isReplaceAll = true for both.
  (insertAt is ignored for both)
  For both objects specified by the value InputPins of A and B, a RemoveAll method call is called. Then, the AddObject method call is generated, taking the two objects specified by the value InputPins as target and parameter. (example 7)

Examples:

1) A CreateLinkAction dealing with two single-valued properties:



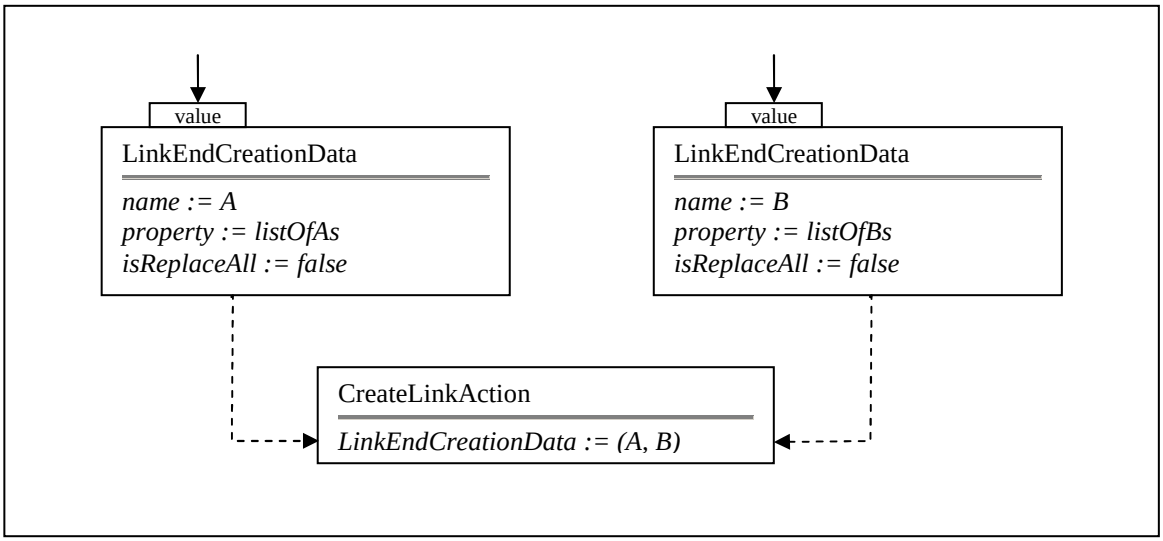
Assumptions:

- the value InputPin of A can be resolved to objectA.
- the value InputPin of B can be resolved to objectB.

This is mapped to the following Java code fragment:

```
objectB.setSingleAttrA(objectA);
```

2) A CreateObjectAction dealing with multiple-valued, ordered properties. For both LinkEndCreationDatas, isReplaceAll is set to false and no insertAt InputPin is contained:



Assumptions:
- the value InputPin of A can be resolved to objectA.
- the value InputPin of B can be resolved to objectB.

This is mapped to the following code fragment:

```
objectB.addToListOfAs(objectA);
```

3) Same setting as example 2, but A has an insertAt InputPin.

Assumptions:
- the value InputPin of A can be resolved to objectA.
- the value InputPin of B can be resolved to objectB.
- the insertAt InputPin of A can be resolved to 3.

This is mapped to the following code fragment:

```
objectB.addAtIndexToListOfAs(3, objectA);
```

4) Same setting as example 2, but both LinkEndDatas have an insertAt InputPin.

Assumptions:
- the value InputPin of A can be resolved to objectA.
- the value InputPin of B can be resolved to objectB.
- the insertAt InputPin of A can be resolved to 3.
- the insertAt InputPin of B can be resolved to 5.

This is mapped to the following code fragment:

```
objectB.addAtIndexToListOfAs(3, objectA, 5);
```

5) Same setting as example 2, but isReplaceAll of A is set to true.

Assumptions:
- the value InputPin of A can be resolved to objectA.
- the value InputPin of B can be resolved to objectB.

This is mapped to the following code fragment:

```
objectB.removeAllFromListOfAs();
objectB.addToListOfAs(objectA);
```

6) Same setting as example 2, but isReplaceAll of A is set to true and B contains an insertAt InputPin.

Assumptions:
- the value InputPin of A can be resolved to objectA.
- the value InputPin of B can be resolved to objectB.
- The insertAt InputPin of B can be resolved to 3.

This is mapped to the following code fragment:

```
objectB.removeAllFromListOfAs();
objectA.addToListOfBs(3, objectB);
```

7) Same setting as example 2, but isReplaceAll is set to true for both A and B:

Assumptions:
- the value InputPin of A can be resolved to objectA.
- the value InputPin of B can be resolved to objectB.

This is mapped to the following code fragment:

```
objectA.removeAllFromListOfBs();
objectB.removeAllFromListOfAs();
objectA.addToListOfBs(objectB);
```

6.3.6.3 DestroyLinkAction

The mapping of a DestroyLinkAction depends on the two associated LinkEndDestructionData elements (in the following called A and B).

If the properties specified by A and B are non-multiple, the DestroyLinkAction is mapped to a Setter method call with the value InputPins of A as target and null as parameter (see example 1). This setter method will set the respective property of B to null.

If the properties specified by A and B are multiple-valued and isOrdered is set to false, the destroyAt InputPin is ignored (the “destroyAt = null” case is addressed in the following mappings).

If the properties specified by A and B are multiple-valued and isUnique is set to false, the property isRemoveDuplicates is ignored (the “isRemoveDuplicates = false” case is addressed in the following mappings).

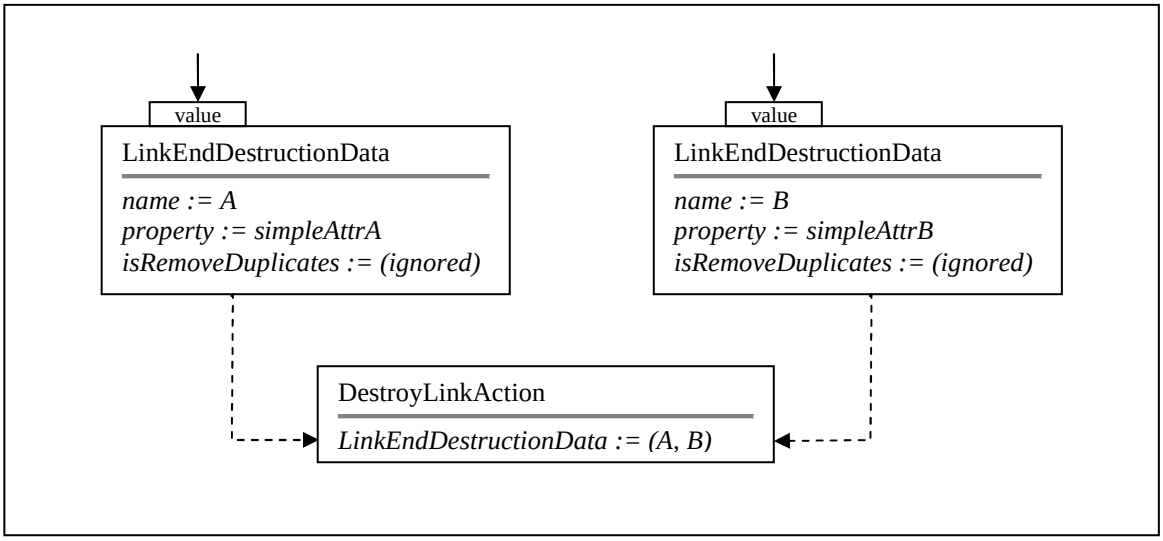
Moreover, the property isRemoveDuplicates of A and B and their optional destroyAt InputPin have to be taken into account:
- isRemoveDuplicates = false for both:
  - destroyAt = null for both:
    An ObjectRemove method call is generated. The target and parameter of this method is specified by the value InputPins of A and B (see example 2).

- destroyAt not null for A, destroyAt = null for B
  A PositionRemove method call is generated. The target is specified by the value InputPin of B, the position parameter by the destroyAt InputPin of A (see example 3).
- destroyAt = null for A, destroyAt not null for B
  (symmetric to the last case)
- destroyAt not null for both:
  A DoublePositionRemove method call is generated. The target is specified by the value InputPin of B, the first position parameter by the insertAt InputPin of A, the second position parameter by the insertAt InputPin of B (example 4).

- isDestroyDuplicates = true for A.
  (destroyAt of A and isDestroyDuplicates and destroyAt of B are ignored, as all Links between A and B are destroyed.)
  An empty while loop with an ObjectRemove method call as test statement is generated. The target of the method call is specified by the value InputPin of B, the parameter by the value InputPin of A. (example 5)
- Other Combinations: Similar to the last case.

Examples:

1) A DestroyLinkAction dealing with simple-valued (non-multiple) properties:

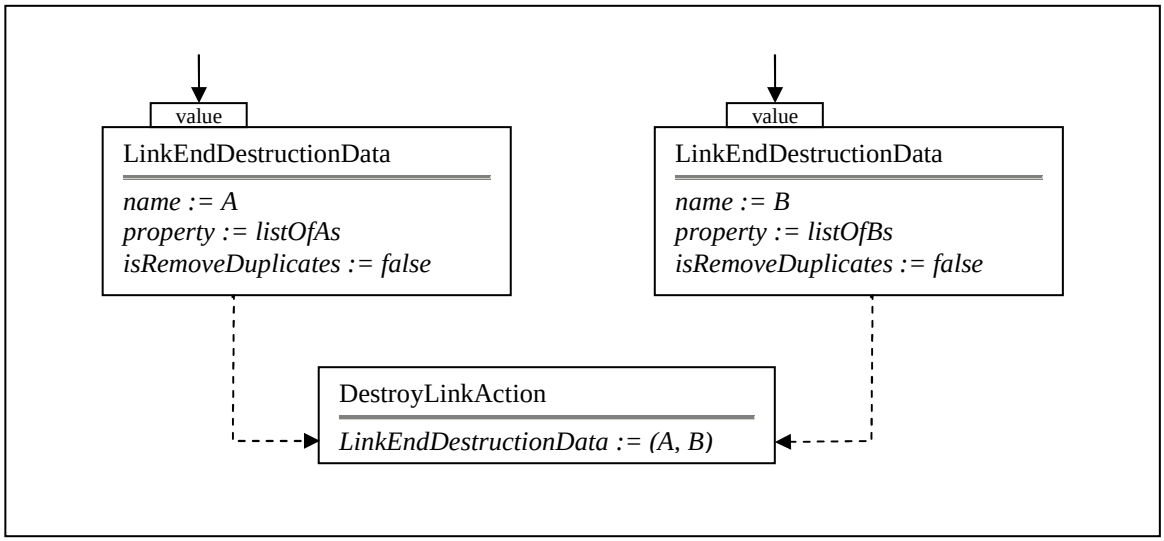


Assumptions:
- the value InputPin of A can be resolved to objectA.
- the value InputPin of B can be resolved to objectB.

This is mapped to the following code fragment:

```
objectB.setSimpleAttrA(null);
```

2) A DestroyLinkAction dealing with multiple-valued, ordered properties. The property isDuplicatesRemove is set to false for both A and B, no destroyAt InputPins are contained:



Assumptions:
- the value InputPin of A can be resolved to objectA.
- the value InputPin of B can be resolved to objectB.

This is mapped to the following code fragment:

```
objectB.removeFromListOfAs(objectA);
```

3) Same setting as example 2, but A has a destroyAt InputPin.

Assumptions:
- the value InputPin of A can be resolved to objectA.
- the value InputPin of B can be resolved to objectB.
- The destroyAt InputPin of A can be resolved to 3.

This is mapped to the following code fragment:

```
objectB.removeAtIndexFromListOfAs(3);
```

4) Same setting as example 2, but both A and B have a destroyAt InputPin.

Assumptions:
- the value InputPin of A can be resolved to objectA.
- the value InputPin of B can be resolved to objectB.
- The destroyAt InputPin of A can be resolved to 3.
- The destroyAt InputPin of B can be resolved to 4.

This is mapped to the following code fragment:

```
objectB.removeAtIndexFromListOfAs(3, 4);
```

5) Same setting as example 2, but isDuplicatesRemove is set to true for A.

Assumptions:
- the value InputPin of A can be resolved to objectA.
- the value InputPin of B can be resolved to objectB.

This is mapped to the following code fragment:

```
while (objectB.removeFromListOfAs(objectA){}
```

6.3.7 ValueProcessingActions

6.3.7.1 ValueSpecificationAction

A ValueSpecificationAction is only used in the context of a Clause (as part of a ConditionalNode) and a LoopNode. In both cases, the purpose is to introduce the expression used as test criterion. The generated code for this action is the Java code generated from the OCLExpression of the contained value input pin.

6.3.7.2 ValueSpecification

In Vide, ValueSpecifications are only used indirectly as superclass of OpaqueExpression, which is again a superclass of ExpressionInOcl. Therefore, it is sufficient to only focus on code generation for OpaqueExpression and for ExpressionInOCL. If we have an OpaqueExpression, then the generated code would be the body. The generation of java code from OCL expressions is completely covered by integration of an external OCL Compiler, consequently we will not address it here.

6.3.8 Variable Actions

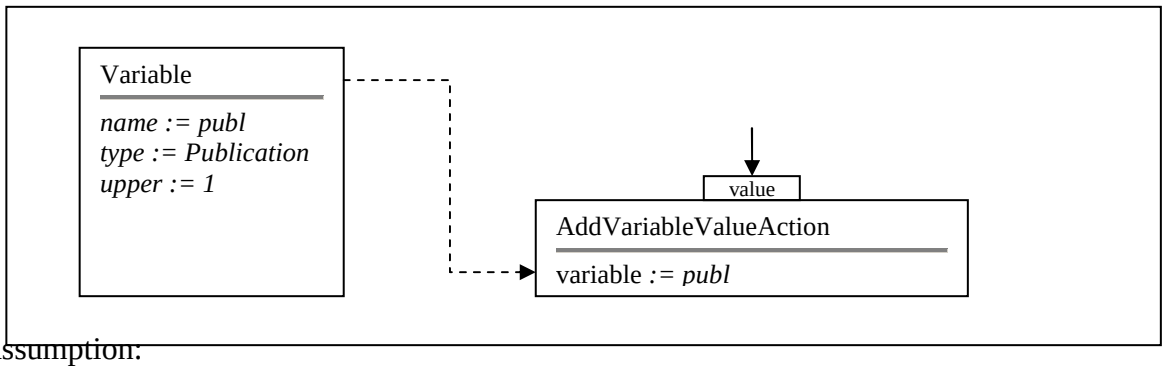
6.3.8.1 AddVariableValueAction

The mapping of this action is very similar to the mapping of the action AddStructuralFeatureValueAction. In the simple case of a non multiple-value variable this action is mapped to an assignment statement in Java (*example 1*). The name of the variable is used in the left side of the assignment. The value input pin is used to generate the expression value at the right side of the assignment. In the case of a multi-value variable several cases have to be differentiated as explained below:

- isReplaceAll = true: generates a replacement assignment (analog to example 1).
- isReplaceAll = false:
    - not ordered: generates an insertion (call to the add method of the Java collection interface List, example 2).
    - ordered:
        - insertAt = null: generates an insertion at the end of the list using the add method of the Java collection interface List, (analog to example 2).
        - insertAt not null: generates an insertion at the position specified by insertAt using the add method of the List interface that takes a position and a value (example 3).

Examples :

1) An AddVariableValueAction with a non-multiple variable:



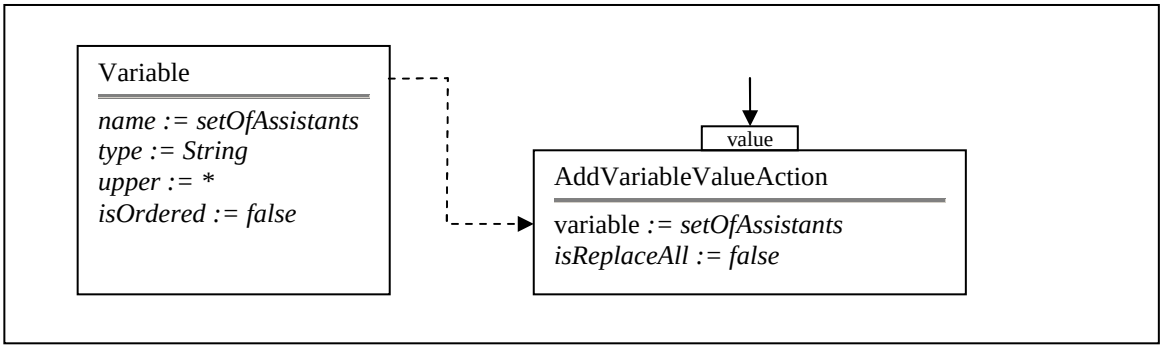
Assumption:

- the value InputPin can be resolved to a Publication *aPublicatio*n

The following code is generated:

```
publ = aPublication;
```

2) An AddVariableValueAction with a multiple-valued, not ordered variable and the property replaceAll is set to false:



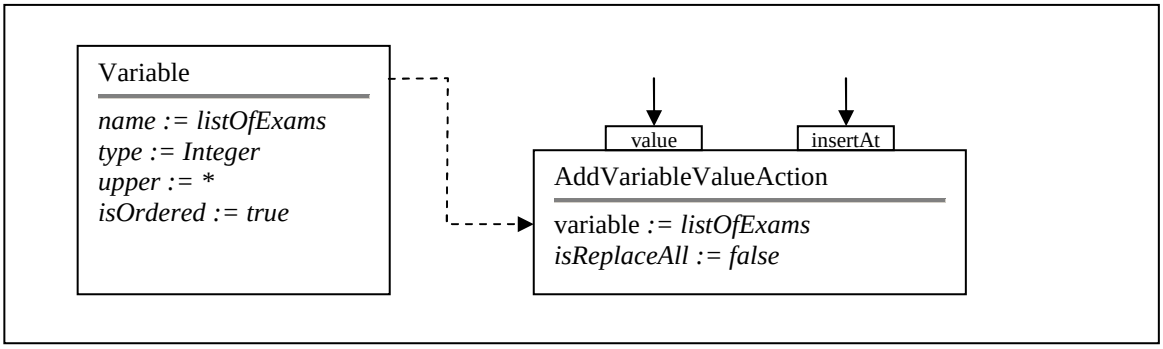
Assumption:

- the value InputPin can be resolved to a String *"Dr. Best"*

The following code is generated:

```
setOfAssistants.add("Dr. Best");
```

3) An AddVariableValueAction with a multiple-valued, ordered variable and the property replaceAll set to false. Additionally, it contains an insertAt InputPin:



Assumption:

- the value InputPin can be resolved to an Integer *"examID"*
- the insertAt InputPin can be resolved to an Integer *"indexPos"*

The following code is generated:

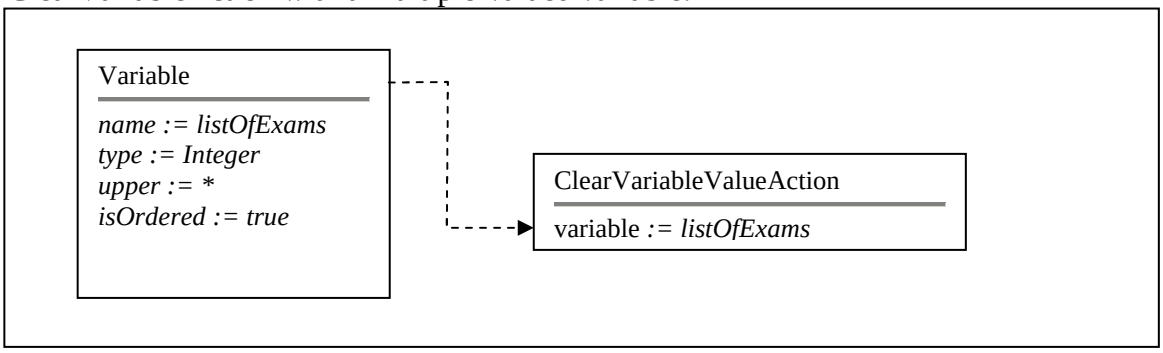
```
listOfExams.add(indexPos, examID);
```

6.3.8.2 ClearVariableValueAction

If case of a single-value variable that has a non-primitive type an assignment of the variable to null is generated (var = null;). In case of a primitive type, the variable is set to the default value (see table in section ClearStructuralFeatureAction). In the case of a multi-value variable the method *clear* of the Java collection classes is called to delete all values contained in the variable.

Example (based on the sample model introduced in Appendix A):

A ClearVariableAction with a multiple-valued variable:



The following code is generated:

```
listOfExams.clear();
```

6.3.8.3 RemoveVariableValueAction

If the Variable is not multi-valued (upper == 1), this action is treated similarly to ClearVariableAction: for primitive types the variable is set to the default value, otherwise the variable is set to null.
If the variable is multi-valued, four cases can occur depending on the properties isUnique and isOrdered:

1) unique and ordered (UniqueList):

isRemoveDuplicates is ignored because the list is already unique.

a) removeAt not null: remove(int)-method of class java.util.List is called. An object possibly specified as value of the value input pin is ignored (see example 1).

b) removeAt = null: remove(Object)-method of class java.util.List is called (see example 2).

2) unique and unordered (Set):

isRemoveDuplicates is ignored because the Set is already unique.

removeAt is ignored because the Set is unordered.

remove(Object)-method of class java.util.Set is called (analog to example 2).

3) not unique and ordered (List):

a) removeAt not null

I. isRemoveDuplicates = true: first, a new, temporary variable (cf. chapter 1.1.4) of the same type as the variable elements is declared and initialized with the variable element at position removeAt. Afterwards, remove(Object)-method of class java.util.List is called as a test statement in an empty while loop. The new variable serves as parameter of the remove method. An object possibly specified as value of the value input pin is ignored. (example 3)

II. isRemoveDuplicates = false: remove(int)-method of class java.util.List is called. An object possibly specified as value of the object input pin is ignored (analog to example 1).

b) removeAt = null

I. isRemoveDuplicates = true: A remove(Object)-method of class java.util.List with the is called as a test statement in an empty while loop. The object specified with the value InputPin is the parameter of this method. (example 4)

II. isRemoveDuplicates = false: remove(Object)-method of the class java.util.List is called (analog to example 2).

4) not unique and unordered (Collection):

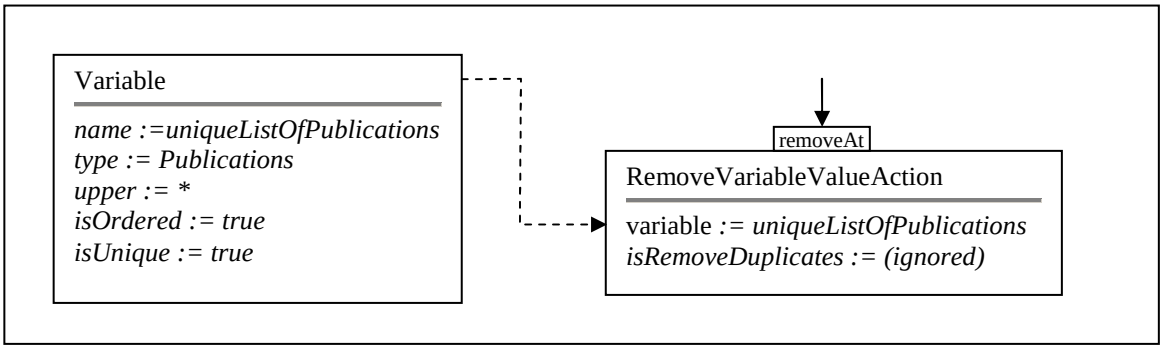
removeAt is ignored because the Collection is unordered.

(Comparable to case 3b).

a) isRemoveDuplicates = true: A remove(Object)-method of class java.util.Collection with the is called as a test statement in an empty while loop. The object specified with the value InputPin is the parameter of this method. (analog to example 4).

b) isRemoveDuplicates = false: remove(Object)-method of the class java.util.Collection is called (analog to example 2).

Examples :



1) RemoveVariableValueAction with an ordered and unique variable. A removeAt InputPin is contained.

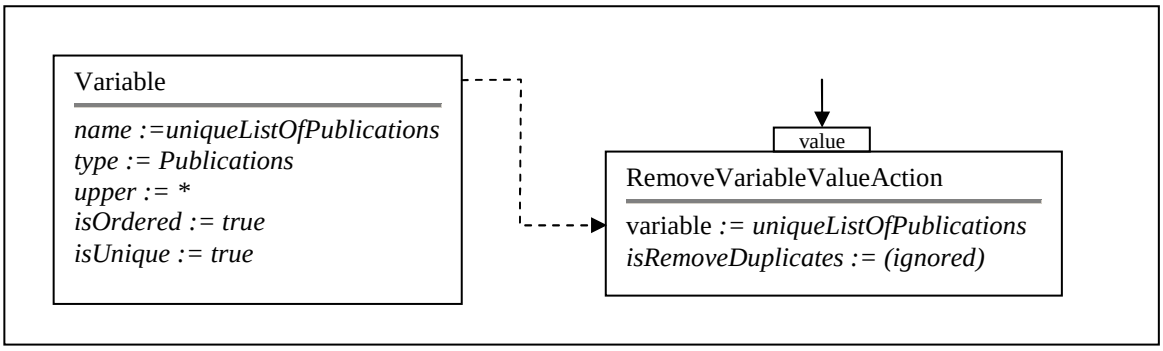
Assumption:

- the removeAt InputPin can be resolved to an Integer *"examID"*

The following code is generated:

```
uniqueListOfPublications.remove(indexPos);
```

2) RemoveVariableValueAction with an ordered and unique variable. A value InputPin, but no removeAt InputPin is contained.



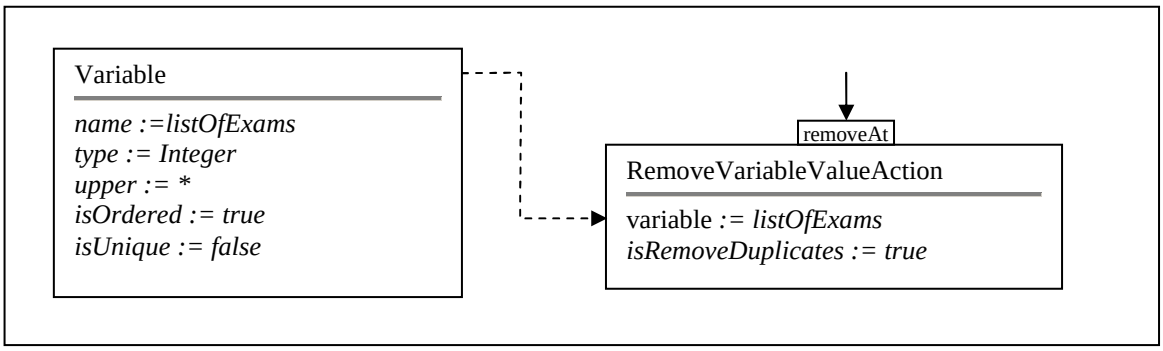
Assumption:

- the value InputPin can be resolved to a Publication *"publ"*

The following code is generated:

```
uniqueListOfPublications.remove(publ);
```

3) RemoveVariableValueAction with an ordered and non-unique variable. The property isRemoveDuplicates is set to true.



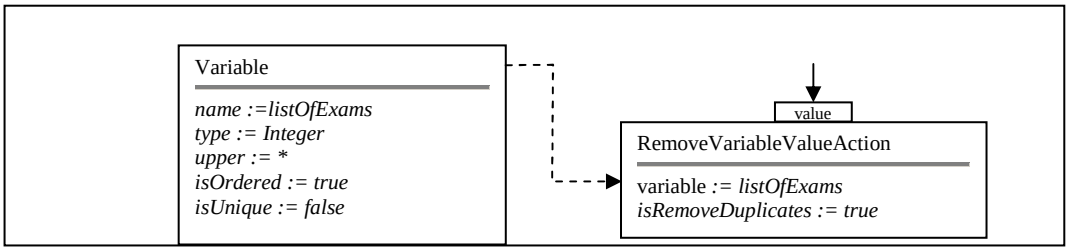
Assumption:

- the removeAt InputPin can be resolved to an Integer *"indexPos"*

The following code is generated:

```
int var__1 = listOfExams.get(indexPos);
while (listOfExams.remove(var__1)){}
```

4) RemoveVariableValueAction with an ordered and non-unique variable. The property isRemoveDuplicates is set to true.



Assumption:

- the value InputPin can be resolved to an Integer *"examID"*

The following code is generated:

```
while (listOfExams.remove(examID)){}
```

6.4 Mapping of Activities to Java

This section presents the mapping of metaclasses defined in the activity package.

The notation used to present mapping is:

- `Courier` font stands for literal expressions
- *Italic* font stands for VIDE metamodel terms (classes, properties)
- map word stands for : apply the mapping of the following metamodel term.

6.4.1 Activity

Activities can be mapped to Java method declaration using their name and their parameter. But this information is more formally specified by the owning operation. Therefore, there is no direct mapping of activities to Java.

6.4.2 ActivityEdge

This is an abstract class and there is no direct mapping to Java.

6.4.3 ActivityNode

This is an abstract class and there is no direct mapping to Java.

6.4.4 Behavior

Although this class is not abstract, there is no direct instance of it. So there is no direct mapping to Java

6.4.5 ConditionalNode & Clause

For the first clause, generate an `if` statement, for other clauses, generate `else if` statements.

The last clause has an always true *test* association by construction so there is no need to do specific mapping for it.

For each clauses, its *test* association is used to generate the Java test.

The order, the clauses are generated, is given by the *successorClause* and *predecessorClause* associations of the *Clause* metaclass.

Example :

```
if  (map  1st test clause) {
     map  1st body clause
}
else if (map  2nd test clause) {
     map 2nd body clause
}
else if (true) {
     map last body clause
}
```

Remarks:

It is assumed that the ExecutableNode in the *body* association are ordered in their sequential execution position.

It is assumed that the ExecutableNode in the *test* association have an empty *handler* association.

It is assumed that *test* association contains only one *ExecutableNode*

VIDE Switch statement are transformed in `if else if` expressions in Java

6.4.6 ControlFlow

ControlFlow is not mapped to specific Java statement but it is important to generate statement in the appropriate sequential order.

6.4.6.1.1.1 ControlNode

This is an abstract class and there is no direct mapping to Java.

6.4.6.1.1.2 ExceptionHandler

It is mapped to Java with the following pattern :

```
try {
     map protectedNode
}
catch (map exceptionType[0] map exceptionInput ) {
     map  handlerBody
}
…
catch (map exceptionType[n] map exceptionInput ) {
     map handlerBody
}
```

remarks : *protectedNode* is the back pointer of *handler* association in ExecutableNode

6.4.6.1.1.3 ExecutableNode

This is an abstract class and there is no direct mapping to Java.

Handler association is checked to map ExceptionHandler.

6.4.7 ExpansionRegion, ExpansionNode

It is mapped to Java 5 with the following pattern:

```
for ( A a : map inputElement[0] )
{
     map StructuredActivityNode
}
```

Where *A* is mapped from *type* association of *inputElement*. Type should be a collection, *A* is the type of elements in the collection.

VIDE doesn't support returning elements from *ExpansionRegion* invocation.

If *Expansionregion* has more than one *inputElement*, the loop is duplicated for every *inputElement*.

6.4.8 ForkNode

It is mapped to a Java thread creation. The outgoing flow is generated inside the `run` method of the thread and the method is finished when the corresponding *JoinNode* is encountered.

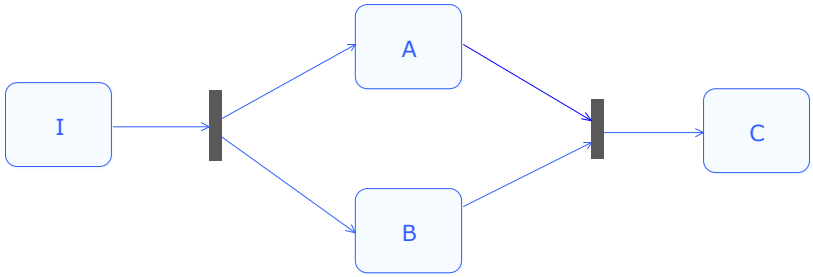


Figure 30 : Example of Forknode

The mapping of the example presented Figure 1 is :

```
map I
Thread t1 = new Thread() {
   public void run() {
       map A
        } // the JoinNode is reached
}.start();
Thread t2 = new Tread() {
   public void run() {
       map B
        } // the JoinNode is reached
}.start();
// wait for both threads to finish
t1.join();
t2.join();
map C
```

6.4.9 LoopNode

It is mapped to `while` or `do while` statement depending of the value of *isTestedFirst* attribute.

The whole Java loop expression is embedded in a block to conceal *setupPart* variables inside the loop perimeter.

- If *isTestedFirst* is true :

```
{
   map setupPart[0]
       ...
   map setupPart[n]
       while( map test[0] ) {
           map bodyPart [0]
   }
```

```
}
```

- If *isTestedFirst* is false :

```
{
    map setupPart[0]
     …
    map setupPart[n]
   do {
      map bodyPart[0]
   } while( map test[0] )
}
```

Remarks

It is assumed that *test* association contains only one *ExecutableNode*

It is assumed that *bodyPart* association contains only one *ExecutableNode*

Vide *for* loops are generated as Java `while` loops.

6.4.10 ObjectFlow

There is no direct mapping of ObjectFlow to Java. ObjectFlow are followed to find variable or parameter to be passed to method invocation or assignment statements.

6.4.11 ObjectNode

This is an abstract class and there is no direct mapping to Java.

6.4.12 SequenceNode

It is mapped to a block with variable declarations and executable node taken respectively from *variable* and *executableNode* associations

```
{
   map variable[0]
   …
   map variable[n]
   map executableNode[0]
   …
   map executableNode[n]
}
```

6.4.13 StructuredActivityNode

This is an abstract class and there is no direct mapping to Java.

6.4.14 Variable

Modeled Variables:
If a Variable is contained in a StructuredActivityNode (e.g. SequenceNode) it is mapped to a variable declaration at the beginning of the generated code block. The reason for this is that the variables are not contained in a special order (or mentioned in the ordered list of ExecutableNodes).

The variable is mapped to Java variable declaration as follows :

```
map type map name = map defaultValue ;
```

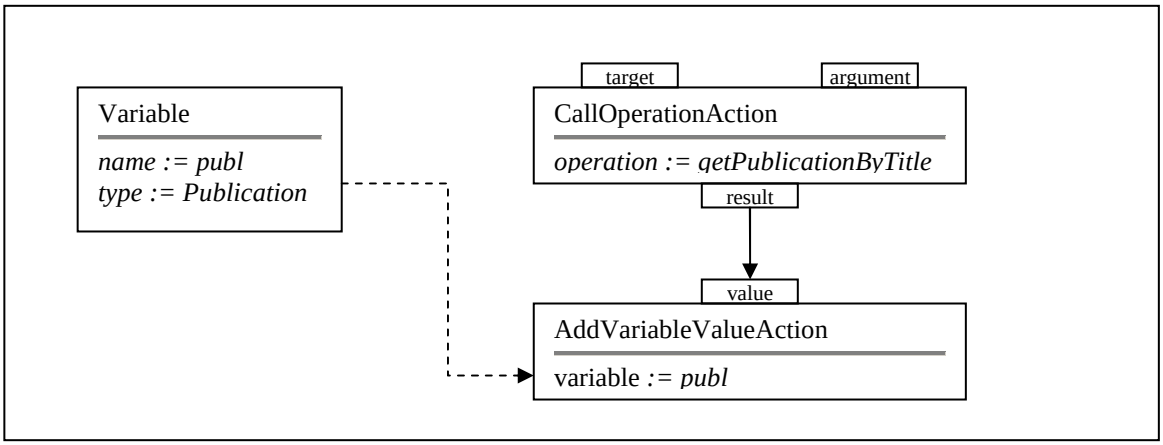
Where *type* association indicates the type of the variable (inherited from TypeElement), *name* is an attribute of *Variable* (inherited from NamedElement) and *defaultValue* association indicates an optional default value.

If *defaultValue* association is null, variable is initialised with a default value according to its type, as described in Table 2.

Temporary Variables:
Additionally, the mappings described in the following chapters suggest to introduce temporary (not modelled) variables at certain points. These variables are used to store the results of the actions.

The type of the temporary variables can be determined from the result specification of the action. The identifier can be determined from the name of the OutputPin. If the name property is not set, a unique identifier has to be chosen, otherwise name conflicts could appear. To ensure the uniqueness of the chosen identifier, the generator has to check, whether or not an identifier called “var__(i)” (with (i) being the integer value of an counter) is already used in the actual context. The check has to take local variables, parameters and fields of the class into account. If a conflict is detected, the counter is incremented and the check is repeated with “var__(i+1)”. Otherwise “var__(i)” is chosen as identifier and the counter is incremented. The counter is reinitialized when entering another activity.

Example (based on the sample model introduced in Appendix A):
In this example, the result of the Operation getPublicationByTitle contained in class Professor is to be assigned to the Variable publ. This is modelled by introducing the variable publ itself and two ExecutableNodes in sequence:



The variable *publ* of type Publication is mapped to a corresponding variable declaration at the beginning of the code generated for the SequenceNode that contains that variable:

```
Publication publ;
```

As already explained in this chapter, a newly declared temporary variable with the unique name *var__1* is used to map the object flow between the two actions :

```
Publication var__1 =
professorXY.getPublicationByTitle(title);
publ = var__1;
```

6.5 Expressions

This section presents the mapping of the VIDE metamodel expression package. These metaclass comes from the OCL metamodel. They always return a value.

The notation used to present mapping is:

- `Courier` font stands for literal expressions
- *Italic* font stands for VIDE metamodel terms (classes, properties)
- map word stands for : apply the mapping of the following metamodel term.

6.5.1 CallExp

This is an abstract class and there is no direct mapping to Java.

6.5.2 FeatureCallExp

This is an abstract class and there is no direct mapping to Java.

6.5.3 IfExp

It is mapped to the Java ternary operator `?:`

`(`map *Condition*`) ?` map *thenExpression* `:` map *elseExpression*

6.5.4 IterateExp

It is generated as an instantiation of an anonymous class. This generation pattern allows using a return statement instead of an assignment of *result*. The context is made available as arguments of the evaluating method.

`ResultType` is the mapping of the type of *result*

```
(new Object() {
   public map result.type eval( map source.type Par,
                                map contextVariable.type self,
                                map parameterVariable.type
                                    parameterVariable.name)
   {
      map result;
      for (map iterator : Par)
      {
         map body;
      }
      return map result.name;
   }
}).eval(map source, this,
        map parameterVariable.representedParameter.name);
```

6.5.5 IteratorExp

This class represents all the predefined VIDE operators that apply on elements of a set. A lot of operators are meaningful for checking the constraints of a model but less useful for processing business logic. Therefore, only the mapping of the more relevant operators is described. In all mappings, the context is made available as arguments of the evaluating method.

- collect

It is mapped to a specialized version of iterateExp, where the results of *body* are added to the result. Note that it is assumed that the body expression refers to the *iterator*.

```
(new Object() {
   public map result.type eval( map source.type Par,
                                map contextVariable.type self,
                                map parameterVariable.type
                                    parameterVariable.name)
   {
      map result;
      for (map iterator : Par)
      {
```

```
        map result.name.add(map body);
      }
      return map result.name;
   }
}).eval(map source, this,
        map parameterVariable.representedParameter.name);
```

- sortedBy

It is mapped to an anonymous class that calls the Java sort primitive defined on Collection and return the List. The comparator required by the sort primitive is also an anonymous class.

```
(new Object() {
   public map result.type eval( map source.type Par,
                                map contextVariable.type self,
                                map parameterVariable.type
                                    parameterVariable.name)
  {
      Collection.sort(Par, new Comparator<source.type.elementType>() {
         public int compare(source.type.elementType o1,
                            source.type.elementType o2) {
            if (o1. map body < o2.map body)
               return -1;
            else if o1. map body = o2.map body)
               return 0;
            else
               return 1;
       } });
     return Par;
  }).eval(map source, this,
          map parameterVariable.representedParameter.name);
```

- select

It is mapped to a specialized version of iterateExp, where elements that satisfy the body are added to the result. Note that it is assumed that the body expression is boolean and refers to the *iterator*.

```
(new Object() {
   public map result.type eval( map source.type Par,
                                map contextVariable.type self,
                                map parameterVariable.type
                                    parameterVariable.name)
   {
      map result;
      for (map iterator : Par)
      {
        if (map body) map result.name.add(map iterator.name);
```

```
      }
      return map result.name;
   }
  }).eval(map source, this,
          map parameterVariable.representedParameter.name);
```

- exists

It is mapped to a specialized version of iterateExp, where as soon as an element satisfies *body* `true` is returned and `false` if no element satisfies it. Note that it is assumed that the *body* expression is boolean and refers to the *iterator*.

```
(new Object() {
   public map result.type eval( map source.type Par,
                                map contextVariable.type self,
                                map parameterVariable.type
                                    parameterVariable.name)
   {
      for (map iterator : Par)
      {
        if (map body) return true;
      }
      return false;
   }
  }).eval(map source, this,
          map parameterVariable.representedParameter.name);
```

- forAll

It is mapped to a specialized version of iterateExp, where as soon as an element doesn't satisfy *body* `false` is returned and `true` if all elements satisfy it. Note that it is assumed that the *body* expression is boolean and refers to the *iterator*.

```
(new Object() {
   public map result.type eval( map source.type Par,
                                map contextVariable.type self,
                                map parameterVariable.type
                                    parameterVariable.name)
   {
      for (map iterator : Par)
      {
        if (! map body) return false;
      }
      return true;
   }
  }).eval(map source, this,
          map parameterVariable.representedParameter.name);
```

6.5.6 LiteralExp

It is directly mapped to its name, inherited from *NamedElement.*

```
name
```

6.5.7 LoopExp

Although not formally abstract class, this class has no direct mapping to Java.

6.5.8 NavigationCallExp

NavigationCallExp is a reference to a Property attached to an Association. It's mapping will differ if it's *qualifier* is empty or not.

No qualifier: it is mapped to the name of its *navigationSource (*Property*)*

```
navigationSource.name
```

With qualifier: the mapping of the qualifier is mapped at the beginning using a dot to separate.

```
map qualifier. navigationSource.name
```

6.5.9 OclExpression

This is an abstract class and there is no direct mapping to Java.

6.5.10 OclVariable

OclVariables are mapped using VariableExps

```
name
```

6.5.11 OpaqueExpression

It is assumed that OpaqueExpression contains only Java code and it is mapped directly to its *body*.

```
Body[1]
```

6.5.12 OperationCallExp

OperationCallExp is used to represent unary operator (not, unary -), binary operators (+, -,*, /, <, >, =, <>, <=, >=, or, xor, and) or user defined operation call. Its mapping depend on that.

Unary Operator

referredOperation.name map *source*

Binary Operator

map *source* *referredOperation.name* map *argument[1]*

User defined operator

map *source. referredOperation.name* (map *argument[1*] , … map *argument[n]*)

6.5.13 PropertyCallExp

PropertyCallExp is used to get the value of an attribute. It is map to the according syntax in java, a dot notation.

map *source. referredProperty.name*

6.5.14 VariableExp

VariableExp is used to get the value of a variable. It is map to the name of the variable.

referredVariable.name

6.5.15 ExpressionInOcl

ExpressionInOcl is the root of an Ocl expression, it is mapped to its *bodyExpression*

map *bodyExpression*

7 VIDE to J2EE

This section describes the mapping of VIDE to two APIs defined in Java EE 5, the last version of J2EE presented section 5.1. These APIs are:

- Java Persistence API (JPA)
- Java Web Services (JAX-WS)

Note : SAP Web Application Server is fully compatible with both JPA and JAX-WS, therefore there is no particularities for this platform in this document. Nevertheless, the Java compiler will be validated on this platform to ensure interoperability of the mapping and the developed compiler.

7.1 Java Persistence API

This chapter presents the mapping of VIDE to the Java Persistence API (JPA). It started with a little presentation of JPA, then, the stereotypes required for JPA are defined before the definition of the mapping for structure, action and expression where mapping to Java Persistence Query Language (JPQL) is presented.

7.1.1 Presentation of JPA

JPA (Java Persistence API) defines an interface to persist normal Java objects (or POJO's in some people terminology) to a datastore. JPA is tightly coupled to RDBMS datastores. JPA is a *standard* approved in June 2006 as part of "EJB3" though can be used outside of the J2EE container. JPA defines the interface that an implementation has to implement. It replaces JDO the previous persistent API specified by Sun.

JPA defines persistent property of the Java classes through the use of Java annotations. This is a clear advantage for generating JPA code from VIDE program because there is no need to generate an xml file like JDO.

JPA defines also a query language JPQL that looks like SQL but work in the name space of the Java program, not in the database name space.

7.1.2 VIDE Mapping to JPA

JPA provides more than 10 Java annotations to be able to define complex mappings between Java classes and databases. For this first mapping of VIDE to JPA, we modestly stay at the level of the proof of concept and we only consider simple Java to database mapping, where each table is represented as a class and each column as an attribute or a foreign key to another table.

The notation used to present mapping is:

- `Courier` font stands for literal expressions
- *Italic* font stands for VIDE metamodel terms (classes, properties)
- map word stands for : apply the mapping of the following metamodel term.

7.1.2.1 VIDE Profile for JPA

With the introduction of persistency, it is essential to be able to distinguish between classes whose instances will be store and retrieve in the database and classes whose instances will be transient.

Therefore the stereotype <<`Persistent>>` defined on classes will indicate classes that should be annotated for JPA persistence.

JPA also require that every persistent class define a primary key. To deal with this constraint, the stereotype <<`Id>>` on attribute is defined.

7.1.2.2 Structure

This section defines the Java annotations generated to use JPA for the structure part of the VIDE metamodel.

7.1.2.2.1 Class

If a class is stereotyped Persistent, then the Java annotation `@Entity` is generated before the class declaration.

```
@Entity
public class Student {
…
}
```

If the persistent class extends an abstract class, the annotation `@MappedSuperclass` is added to the super class.

```
@MappedSuperclass
public abstract class Person {
…
}

@Entity
Class Student extends Person {
…
}
```

7.1.2.2.2 Property (Attribute)

If a Property that acts as an attribute of a class is stereotyped Id, then the Java annotation `@Id` is generated before the attribute declaration.

```
@Entity
public class Student {
```

```
@Id
public int id;
…
}
```

7.1.2.2.3 Property (Association)

The annotations generated depend on the multiplicity and the nature of the association.

Figure 31 : Simple composition

The simple composition as presented Figure 31 will generate:

```
@Entity
public class A {

@OneToOne(cascade=CascadeType.ALL, fetch=FetchType.LAZY)
public B b;
```

The annotation and its property indicates that when a A is saved or destroyed, its corresponding B should be (CascadeType.ALL) and FetchType says that B should be loaded from the data store only when the attribute b is read.

Figure 32 : Composition to many

The case presented Figure 32 will generate:

```
@Entity
public class A {

@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.LAZY)
public B b;
```

The annotation becomes OneToMany. The properties have the same meaning than previously.

Figure 33 : Association Many to One

The case presented Figure 33 will generate:

```
@Entity
public class A {

@ManyToOne(fetch=FetchType.LAZY)
public B b;
```

The annotation is ManyToOne because several A can be linked to the same B. The cascading is useless because the association is not a composition.

If `b` multiplicity is `*` then the annotation is `ManyToMany` with the same property.

Figure 34 : Bidirectional association

The case presented Figure 34 will generate:

```
@Entity
public class A {

@ManyToOne(fetch=FetchType.LAZY)
public B b;

…

@Entity
public class B {

@OneToMany(mappedBy=A.b)
public A a;
```

There is no modification in A class, but in class B, mappedBy property tells that the association ‘belongs’ to class A and that `a` attribute will be loaded correctly.

7.1.2.3 Activity

Mapping VIDE to JPA doesn't affect the Activity metaclasses but some code to declare the entity manager and transactions is required to use JPA. For this first version, we propose to declare the entity manager as a global singleton and to gather all database operations as a single transaction. This code is added in the `main` operation.

```
public class VIDEEntityManager {
     public static EntityManager em;
    public static EntityManager getEM()
    {
       if (em == null)
       {
           em =
Persistence.createEntityManagerFactory("default").createEntityManager();
       }
       return em;
    }
}
```

```
public class VIDEApp {
    public static void main (String[] args){
       VIDEEntityManager.getEM().getTransaction.begin();
       // generated code here
       VIDEEntityManager.getEM().getTransaction.commit();
    }
}
```

7.1.2.4 Actions

Only *CreateObjectAction* and *DestroyObjectAction* are impacted.

7.1.2.4.1 CreateObjectAction

Apart from creating the new object, it has to be made persistent, the generated code is :

```
   Professor p = new Professor();
   VIDEEntityManager.getEM().persist(p);
```

7.1.2.4.2 DestroyObjectAction

The object should be deleted in the database.

```
   VIDEEntityManager.getEM().remove(p);
   P = null;
```

7.1.2.5 Expression

JPA give us the opportunity to use OCL expression to make queries on the database.

How to decide to generate a query or a simple mapping described chapter 6.5 ?

If the *source* of an OCL expression refer to a classifier (VIDE allows to skip the *allInstances* primitive of OCL) then it is a query, if it is a property or a variable then it is a simple mapping.

In the following sections `em` is used for `VIDEEntityManager.getEM()`.

7.1.2.5.1 Trivial mapping

This mapping consists to retrieve all the persistant instances of a class without any selection criterion and pass this list to the following expression.

```
em.createQuery("from source").getResultList()
```

This mapping is used for `exist` and `forall`. `collect`, `sortedBy` and `select` can take more advantages of JPQL.

7.1.2.5.2 Mapping collect to JPA query

The *body* association is used in the `select` clause.

```
em.createQuery("select map iterator.map body from source map
iterator").getResultList()
```

7.1.2.5.3 Mapping sortedBy to JPA query

The *body* association is used in the `sortby` clause.

```
em.createQuery("from source map iterator group by map
body").getResultList()
```

This mapping requires that *body* is a *PropertyCallExp*, because `group by` clause is defined only on attribute in JPQL. If not, a trivial mapping should be generated.

7.1.2.5.4 Mapping select to JPA query

The *body* association is used in the `where` clause.

```
em.createQuery("from source map iterator where map body").getResultList()
```

7.2 Web services

This chapter describes the mapping of the VIDE Web Services profile to J2EE and focuses on the implementation of that mapping in the context of the Java/J2EE model compiler. Thereby, two use cases are considered. In the first, one a VIDE class is published as a Web Service an appropriate Java code should be generated whereas in the second case an external Web Service is consumed from within VIDE class.

The main idea in the first case is to generate Java API for XML Web Services (JAX-WS) annotations in the Java code. Since, the implementation of the consumed Web Service operation is not available; the model compiler should generate code that calls appropriate client-side Web Service proxies rather then generating Java code from UML actions.

The structure of this document is as follows. Section 7.2.1 presents the VIDE profile for Web Services. Section 7.2.2 presents the Web Service support in the target platform and mainly the JAX-WS annotations. Section 7.2.3 explains how to VIDE model compiler to J2EE implements the publishing of a VIDE class as a Web Service and Section 7.2.4 explains how it implements the consumption of an external Web Service.

7.2.1 VIDE Web Services Profile

In the following, the stereotypes of the VIDE Web Service profile are described.

- **Stereotype ConsumedService**

This stereotype designates that a class will be a proxy to a remote web service conforming to a certain WSDL contract. The operations of that class are associated to remote calls to the operations of a given web service. Because of that, elements marked with this stereotype cannot be attached any OCL code to their body.

This metaclass does not exist in UML metamodel and is implemented as a stereotype applied to Class.

Generalizations
Class
Attributes
URL:String[1] – WSDL contract address
portType:String[1] – represented interface

- **Stereotype PublishedService**

Applicable to class without any attributes defined. This stereotype indicates that a class should be exposed as a web service endpoint. Those are assumed to be automatically started at the beginning of model execution.

Generalizations
Class
Attributes:

Namespace:String[1] - defaults to (filtered) containing package global name

- **Stereotype PublishedOperation**

Marks those operations, which should be available as operations of the published Web Service. They can be applied only when the containing class is marked with publishedService. All types used as input or output parameters such operations are mapped to XSD types definition of types WSDL section.

Generalizations
Class

7.2.2 Java Web Service annotations

Fortunately, Web Services in the J2EE platform is based on annotations, which makes mapping VIDE PIM stereotypes to J2EE simple as an appropriate annotation has to be generated (and not methods).

In the following, we present the JAX-WS annotations that are relevant for mapping the VIDE Web Service profile to J2EE.

- **javax.jws.WebService** The purpose of this annotation is to mark an endpoint implementation as implementing a web service or to mark that a service endpoint interface as defining a web service interface.

Properties:
name: The name of the `wsdl:portType`
targetNamespace: The XML namespace of the WSDL and some of the XML elements generated from this web service. Most of the XML elements will be in in the namespace according to the JAXB mapping rules.
serviceName: The Service name of the web service (`wsdl:service`)
endpointInterface: The qualified name of the service endpoint interface.
portName: The `wsdl:portName`

- **javax.jws.WebMethod** The purpose of this annotation is to expose a method as a web service operation.

Properties:
operationName: The name of the `wsdl:operation` matching this method.
action: The XML namespace of the WSDL and some of the XML elements generated from this web service.
exclude: Used to exclude a method from the Web Service.

These two annotations are the most relevant one for mapping VIDE Web Services profile to Java. The annotations **javax.jws.WebParam** and **WebResult** are also related to our work but they are not necessarily needed.

7.2.3 Publishing a VIDE class as a Web Service

The user may take a state-of-the-art UML class diagrams editor and load the VIDE profiles for Web Services. Then, he can add the stereotype <<publishedService>> to the classes that should be exposed in the Web Service. To select specific methods for exposition in the Web Service, the user may use the stereotype <<publishedMethod>>.

The model compiler from VIDE to J2EE takes the profiled model and generates JAX-WS annotations in the Java class accordingly. The generated source code files have then to be compiled by the user and deployed to a J2EE application server.

An example of the code generated by the VIDE model compiler to make a class exposed as a web service is shown below. The java class is annotated with the JAX-WS annotation @WebService

```
@WebService
public class Opportunity
{
      public float getValue (String curr)
      {
      return this.value();
      }
}
```

To configure which methods of the Java class should be exposed in the Web Service the annotation @WebMethod is generated. This annotation has properties such as operationName, which can be use to give the Web Service operation a different name than that of the class method.

```
@WebService(name="OpportunityService")
public class Opportunity{

      @WebMethod(operationName="getOpportunityValue")
      public float getValue (String currency)
      {
      return this.value();
      }
}
```

The deployment process is beyond the scope of this document and will be described in D9.2.

7.2.4 Consuming an External Web Service

PJIIT is working on WSDL to VIDE import. That is, they will provide a tool for text-to-Model transformation that generates a VIDE model out of the WSDL file. This tool creates a proxy class in the VIDE model for the Web Service and marks it with the <<consumedService>> stereotype.

The stereotype <<consumedService>> has a string property called URL, which stores the URL of the WSDL file of the consumed Web Service. For each such class, the model

compiler generates a Java class, whose methods redirect all calls to operations of the Java Web Service proxy class. For example, assume that the class with the stereotype <<consumedService>> has a method called `wsoperation`. The Java method generated by the model compiler gets a reference to the Web Service proxy class (in the example *ServiceMyPortType*) and then call the same operation on that proxy and passes its parameters to it as shown below.

```
Public int wsoperation (int param1, int parm 2)
{
       //get reference to the local WS port proxy
       ServiceMyPortType port = new Service.getMyPort() ;
       //redirect the call and return result if applicable
       return port.wsoperation(param1,param2);
}
```

The generated code will only work correctly if the client-side Web Service proxy is available. For that reason, the model compiler uses the tool *wsimport* and passes the *URL* property of the <<consumedService>> stereotype as parameter.

```
wsimport http://company.com/OpportunityService?wsdl
```

8 The model compiler to ODRA

This chapter presents the specification of the mapping from VIDE metamodel to ODRA code.

8.1 Introduction

The model compiler to ODRA is specified as a function *Map2ODRA*, which maps instances of VIDE metamodel to textual code in SBQL to be executed by the ODRA database system. The definition of this function uses the structural recursion, i.e. the mapping of each kind of nodes is described using the mapping of its subordinate kinds. Fragments of the textual output are marked with red colour to additionally distinguish them from the mapping function invocations.

We will also use a generalization of the function *Map2ODRA* which will apply to a sequence a items. The function *Map2ODRA** has two arguments: a node and an optional separator. The result of *Map2ODRA**(*seq*, *sep*) is the concatenation of the results of the function *Map2ODRA* applied to all elements of the sequence *seq* separated by the separator *sep*. The result of *Map2ODRA**(*seq*) is just the concatenation of the results of the function *Map2ODRA* applied to all elements of the sequence *seq*.

8.2 Structures

8.2.1 Mapping

8.2.1.1 Type hierarchy

8.2.1.1.1 BagType

BagType is mapped to the ODRA system as declaration of multiple elements with cardinality [0..*]. The attribute **elementType** is inherited by the BagType from its generalization CollectionType.

Mapped node *x*	**Mapping result *Map2ODRA*(*x*)**
BagType	*Map2ODRA*(*x*.elementType) [0..*]

8.2.1.1.2 Classifier

In VIDE classifier is just an abstract super-class for data type, association and class. The mapping of a Classifier is defined by its concrete subclasses.

8.2.1.1.3 Class

Class is mapped to ODRA class whose name is obtained by adding the Class suffix:

Mapped node *x*	**Mapping result *Map2ODRA*(*x*)**
Class	*x*.nameClass

For class declaration – see the section “Features of classes” below.

8.2.1.1.4 CollectionType

The mapping of a CollectionType is defined by its concrete subclasses.

8.2.1.1.5 DataType

The mapping of a DataType is defined by its concrete subclasses.

8.2.1.1.6 Enumeration

An enumeration is a data type whose values are enumerated in the model as enumeration literals. The enumerations are not implemented in ODRA. Thus, Enumeration is mapped to string.

Mapped node *x*	**Mapping result *Map2ODRA*(*x*)**
Enumeration	string

8.2.1.1.7 OrderedSetType

OrderedSetType is a collection type constructor that describes a set of elements where each distinct element occurs only once in the set. The ordered set is currently not implemented in ODRA. The OrderedSetType is mapped the same way as BagType.

Mapped node *x*	**Mapping result *Map2ODRA*(*x*)**
OrderedSetType	*Map2ODRA*(*x*.elementType) [0..*]

8.2.1.1.8 PrimitiveType

The PrimitiveType is mapped simply to its name:

Mapped node *x*	**Mapping result *Map2ODRA*(*x*)**
PrimitiveType	*x*.name

8.2.1.1.9 SequenceType

SequenceType is a collection type constructor that describes a list of elements where each element may occur multiple times in the sequence. The sequence is currently not implemented in ODRA. The SequenceType is mapped the same way as BagType.

Mapped node *x*	**Mapping result *Map2ODRA*(*x*)**
SequenceType	*Map2ODRA*(*x*.elementType) [0..*]

8.2.1.1.10 SetType

SetType is a collection type constructor that describes a set of elements where each distinct element occurs only once in the set. The set is currently not implemented in ODRA. The SequenceType is mapped the same way as BagType.

Mapped node *x*	**Mapping result *Map2ODRA*(*x*)**
SetType	*Map2ODRA*(*x*.elementType) [0..*]

8.2.1.1.11 TupleType

TupleType (informally known as record type or struct) combines different types into a single aggregate type. The parts of a TupleType are described by its attributes, each having a name and a type. TupleType is mapped to an ODRA record:

Mapped node *x*	**Mapping result *Map2ODRA*(*x*)**

TupleType	record {
	*Map2ODRA**(*x*.ownedProperty, ;)
	}

8.2.1.1.12 Type

The mapping of a Type is defined by its concrete subclasses.

8.2.1.1.13 VoidType

VoidType represents a type that conforms to all types. The void type is mapped to an empty string.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
VoidType	Empty string

8.2.1.2 Features of classes

Each declared class is mapped to an ODRA declaration of class. Together with this declaration a variable holding the extent of the Class is created. The ODRA class name has suffix `Class`. The extent however has the same name as the VIDE class.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)

Class (declaration)	class *x*.nameClass extends *Map2ODRA**(*x*.superClass, ,) {
	instance *x*.name {
	*Map2ODRA**(*x*.ownedAttribute, ;) ;
	}
	*Map2ODRA**(*x*.ownedOperation)
	}
	
	x.name : *x*.nameClass[0..*]

If the list of super-classes is empty, the phrase extends is omitted.

There is also a special case: when the mapped class has the «module» stereotype, then it has to have the name as the owning package (otherwise an error is reported). In this case, the content of such a class is mapped as directly owned by the ODRA module.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
Class («module» with the same name as its owning package)	*Map2ODRA**(*x*.ownedAttribute, ;) ; *Map2ODRA**(*x*.ownedOperation)

8.2.1.2.1 Association

An Association is not directly mapped to ODRA. It is mapped indirectly through the properties owned by classes.

8.2.1.2.2 BehavioralFeature

A behavioural feature specifies that an instance of a classifier will respond to a designated request by invoking a behaviour. It is mapped to ODRA method declaration. The **raisedException**s are not mapped since this part of a method header is not implemented in ODRA.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)

BehavioralFeature (none of ownerParameter is return)	*x*.name (*Map2ODRA**(*x*.ownedParameter, ;)) {
	Map2ODRA(*x*.method)
	}

BehavioralFeature
(one of ownerParameter is return;
say the *i*-th is a return;
the type of this return parameter is not a class)
	x.name (*Map2ODRA**(*x*.ownedParameter, ;)) :
	Map2ODRA (*x*.ownerParamater[*i*].type)
	{
	Map2ODRA(*x*.method)
	}

BehavioralFeature
(one of ownerParameter is return;
say the *i*-th is a return;
the type of this return parameter is a class)
	x.name (*Map2ODRA**(*x*.ownedParameter, ;)) :
	ref *Map2ODRA* (*x*.ownerParamater[*i*].type)
	{
	Map2ODRA(*x*.method)
	}

8.2.1.2.3 Constraint

A constraint is a condition or restriction expressed in natural language text or in a machine readable language for the purpose of declaring the contract of an element. A Constraint is mapped to an empty ODRA string, since it has nothing to do with execution.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
Constraint	Empty string

8.2.1.2.4 Element

The mapping of an Element is defined by its concrete subclasses.

8.2.1.2.5 Feature

The mapping of a Feature is defined by its concrete subclasses.

8.2.1.2.6 MultiplicityElement

A MultiplicityElement is an abstract metaclass that includes attributes for defining the bounds of a multiplicity. It is mapped to ODRA cardinality declaration. Other attributes are not mapped since ODRA does not implement set or sequence yet.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
MultiplicityElement	[*x*.lower .. *x*.upper]

8.2.1.2.7 NamedElement

The mapping of a Feature is defined by its concrete subclasses.

8.2.1.2.8 Namespace

The mapping of a Feature is defined by its concrete subclasses.

8.2.1.2.9 Operation

An operation inherits its mapping from BehavioralFeature.

8.2.1.2.10 Package

A package is used to group elements, and provides a common namespace for the grouped elements. A package is mapped onto an ODRA module:

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
Package	add module *x*.name { *Map2ODRA**(*x*.ownedType) *Map2ODRA**(*x*.nestedPackage)

}

8.2.1.2.11 PackageableElement

The mapping of a PackageableElement is defined by its concrete subclasses.

8.2.1.2.12 PackageImport

A package import is a relationship that allows the use of unqualified names to refer to package members from other namespaces. A package import is directly mapped to ODRA module import.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
PackageImport	import *x*.importedPackage.name ;

8.2.1.2.13 Parameter

A parameter specifies how arguments are passed into or out of an invocation of an operation. Each Parameter is mapped to an ODRA method parameter. Parameters of class types are always mapped to call-by-reference. Parameters of non-class types are mapped to call-by-reference if they are **out** or **inout**. Otherwise, they are mapped to call-by-value. The **return** output parameter is mapped in a special way (see the mapping of BehavioralFeature), so here its is mapped to an empty string.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
Parameter (class type, direction ≠ **return**)	*x*.name : ref *x*.type.nameClass
Parameter (non class type, direction = **in**)	*x*.name : *Map2ODRA*(*x*.type)
Parameter (non class type, direction ∈ {**inout**, **out**})	*x*.name : ref *Map2ODRA*(*x*.type)
Parameter (direction = **return**)	Empty string

8.2.1.2.14 ParameterDirectionKind

ParameterDirectionKind is not mapped directly. Its mapping is quite indirect defined above together with the mapping of a Parameter. Literally, ParameterDirectionKind is mapped to an empty string.

8.2.1.2.15 Property

A Property is a structural feature. It is mapped onto an ODRA field declaration. The mapping is different for bi-directional association ends. In this case, the ODRA field declaration contains the indication of the reverse relationship.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
Property (association not set, default value not set)	*x*.name : *Map2ODRA*(*x*.type) ;
Property (association not set,	*x*.name : *Map2ODRA*(*x*.type) := *Map2ODRA*(*x*.defaultValue) ;

default value set)	
Property (association set but unidirectional)	*x*.name : ref *Map2ODRA(x*.type) ;
Property (bi-directional association set)	*x*.name : ref *x*.type.name reverse *x*.association.memberEnd->select(*y*|*y*≠*x*).name ;

8.2.1.2.16 RedefinableElement

The mapping of a RedefinableElement is defined by its concrete subclasses.

8.2.1.2.17 Relationship

The mapping of a Relationship is defined by its concrete subclasses.

8.2.1.2.18 StructuralFeature

The mapping of a StructuralFeature is defined by its concrete subclasses.

8.2.1.2.19 TypedElement

The mapping of a TypedElement is defined by its concrete subclasses.

8.2.1.2.20 VisibilityKind

A VisibilityKind is mapped to an empty ODRA string, since visibilities are not implemented in ODRA.

Mapped node *x*	**Mapping result *Map2ODRA*(*x*)**
VisibilityKind	Empty string

8.2.1.3 Services

In order to include Web Services definition and usage from the level of VIDE models we extend the metamodel with three metaclasses. However, Web services are not anyhow marked distinct in ODRA database schema. Therefore all the nodes described above are mapped as ordinary ODRA objects.

The actual deployment of published service interfaces and consumed service proxies is performed with additional commands that are described – together with some general considerations on Web service mapping in section 8.6.

8.2.1.4 Module

Module is a class that is a specialization of a normal Class but has one important difference. It is immediately instantiated after the system start as a singleton object. Module-stereotyped class is required to have the same name as its containing package and is not allowed to be a member of associations. A Module *x* is mapped almost the same way as its generalization's, i.e. Class (see the Class mapping section for the description of this special case).

8.3 Actions

8.3.1 Mapping

8.3.1.1 General Concepts

8.3.1.1.1 Action

An action is a named element that is the fundamental unit of executable functionality. The mapping of an Action is defined by its concrete subclasses.

8.3.1.1.2 InputPin

An input pin is a pin that holds input values to be consumed by an action. The mapping of an InputPin is either defined by one of its subclass (if the InputPin in fact belongs to a subclass) or is equal to mapping of an OutputPin which is the source of an ObjectFlow whose target is this InputPin.

8.3.1.1.3 OutputPin

An output pin is a pin that holds output values produced by an action. The mapping of an OutputPin is defined by the action who owns this OutputPin.

8.3.1.1.4 Pin

A pin is a typed element and multiplicity element that provides values to actions and accepts result values from them. The mapping of a Pin is defined by its concrete subclasses.

8.3.1.1.5 ValuePin

A value pin is an input pin that provides a value to an action that does not come from an incoming object flow edge. The mapping of a ValuePin is just the mapping of the provided ValueSpecification.

Mapped node *x*	**Mapping result *Map2ODRA*(*x*)**
ValuePin	*Map2ODRA*(*x*.value)

8.3.1.2 Invocation Actions

8.3.1.2.1 InvocationAction

Invocation is an abstract class for the various actions that invoke behaviour. The mapping of a InvocationAction is defined by its concrete subclasses.

8.3.1.2.2 CallAction

CallAction is an abstract class for actions that invoke behaviour and receive return values. The mapping of a CallAction is defined by its concrete subclasses.

8.3.1.2.3 CallOperationAction

CallOperationAction is an action that transmits an operation call request to the target object, where it may cause the invocation of associated behaviour. It is mapped an ODRA method call on the target of CallOperationAction.

Mapped node *x*	**Mapping result *Map2ODRA*(*x*)**
CallOperationAction	*Map2ODRA*(*x*.target)

.   *x*.operation.name(*Map2ODRA**(*x*.argument), ,)

8.3.1.2.4 RaiseExceptionAction

RaiseExceptionAction is an action that causes an exception to occur. It is mapped to an ODRA throw statement.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
RaiseExceptionAction	**throw** *Map2ODRA*(*x*.exception) **;**

8.3.1.2.5 ReplyAction

ReplyAction is an action that accepts a set of return values. It is mapped to an ODRA return statement.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
ReplyAction	**return** *Map2ODRA*(*x*.replyValue) **;**

8.3.1.3 Object Actions

8.3.1.3.1 CreateObjectAction

CreateObjectAction is an action that creates an object that conforms to a statically specified classifier and puts it on an output pin at runtime. It is mapped to the ODRA create statement.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
CreateObjectAction	**create** *x*.classifier.name **() ;**

8.3.1.3.2 DestroyObjectAction

This action destroys the object on its input pin at runtime. It is mapped to the ODRA create statement.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
DestroyObjectAction	**delete** *Map2ODRA*(*x*.target) **;**

8.3.1.4 Structural Feature Actions

8.3.1.4.1 AddStructuralFeatureValueAction

AddStructuralFeatureValueAction is a write structural feature action for adding values to a structural feature. It is mapped to an ODRA assignment statement (if isReplaceAll=true) or to an ODRA insert-copy statement (if isReplaceAll=false).

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
AddStructuralFeatureValueAction (isReplaceAll=true; *x*.structuralFeature is not of class type)	*x*.object **.** *x*.structuralFeature.name **:=** *x*.value **;**
AddStructuralFeatureValueAction (isReplaceAll=true; *x*.structuralFeature is of class type)	*x*.object **.** *x*.structuralFeature.name **:=** **ref(** *x*.value **);**
AddStructuralFeatureValueAction (isReplaceAll=false; *x*.structuralFeature is not of class type)	*x*.object **:<<** **(***x*.value **as** *x*.structuralFeature.name**) ;**
AddStructuralFeatureValueAction	*x*.object

(isReplaceAll=false; *x*.structuralFeature is of class type)	**:<** **ref(***x*.value**) ;**

8.3.1.4.2 ClearStructuralFeatureValueAction

ClearStructuralFeatureAction is a structural feature action that removes all values of a structural feature. It is mapped to the ODRA delete statement.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
ClearStructuralFeatureAction	**delete** *x*.object **.** *x*.structuralFeature.name **;**

8.3.1.4.3 RemoveStructuralFeatureValueAction

RemoveStructuralFeatureValueAction is a write structural feature action that removes values from structural features. It is mapped to the ODRA delete statement.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
RemoveStructuralFeatureValueAction	**delete** *x*.object **.** *x*.structuralFeature.name**[***x*.removeAt**];**

8.3.1.4.4 StructuralFeatureAction

StructuralFeatureAction is an abstract class for all structural feature actions. The mapping of StructuralFeatureAction to ODRA is defined by its concrete subclasses.

8.3.1.4.5 WriteStructuralFeatureAction

WriteStructuralFeatureAction is an abstract class for structural feature actions that change structural feature values. The mapping of WriteStructuralFeatureAction to ODRA is defined by its concrete subclasses.

8.3.1.5 Link Actions

8.3.1.5.1 ClearAssociationAction

ClearAssociationAction is an action that destroys all links of an association in which a particular object participates. It is mapped to the ODRA delete statement. The endData is inherited from LinkAction.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
ClearAssociationAction	**delete** *x*.object **.** *x*.endData.name**;**

8.3.1.5.2 CreateLinkAction

This action can be used to create links and link objects. CreateLinkAction is mapped to the ODRA insert-copy statement.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
CreateLinkAction	*x*.inputValue [1] **:<<** **(ref(***x*.inputValue[2]**) as** *x*.endData[1].property.name**) ;**

8.3.1.5.3 DestroyLinkAction

This action destroys a link. DestroyLinkAction is mapped to the ODRA delete statement.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
DestroyLinkAction	**delete**

```
                    (x.inputValue [1] . x.endData[1].property.name as FreshVar
                                                    where
                                        FreshVar = x.inputValue[2]) ;
```

FreshVar is a new (fresh) variable name generated in such a way that it does not occur anywhere in the generated code.

8.3.1.5.4 LinkAction

LinkAction is an abstract class for all link actions that identify their links by the objects at the ends of the links and by the qualifiers at ends of the links. The mapping of LinkAction to ODRA is defined by its concrete subclasses.

8.3.1.5.5 LinkEndCreationData

LinkEndCreationData is not an action. It is not directly mapped to ODRA code. It is used in the mapping of owning LinkAction.

8.3.1.5.6 LinkEndData

LinkEndData is not an action. It is an element that identifies links. It identifies one end of a link to be read or written by the children of LinkAction. It is not directly mapped to ODRA code. It is used in the mapping of owning LinkAction.

8.3.1.5.7 LinkEndDestructionData

LinkEndDestructionData is not an action. It is an element that identifies links. It identifies one end of a link to be destroyed by DestroyLinkAction. It is not directly mapped to ODRA code. It is used in the mapping of owning LinkAction.

8.3.1.5.8 WriteLinkAction

WriteLinkAction is an abstract class for link actions that create and destroy links. The mapping of LinkAction to ODRA is defined by its concrete subclasses. The mapping of WriteLinkAction to ODRA is defined by its concrete subclasses.

8.3.1.6 Value Processing Actions

8.3.1.6.1 ValueSpecification

A value specification is the specification of a (possibly empty) set of instances, including both objects and data values. Its mapping is defined by its concrete subclasses.

8.3.1.6.2 ValueSpecificationAction

ValueSpecificationAction is an action that evaluates a value specification. Its mapping to ODRA code is equivalent to the mapping of the specified value.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
ValueSpecificationAction	*Map2ODRA*(*x*.value)

The mapping of its output pin *result* is the same as mapping of the action.

8.3.1.7 Variable Actions

8.3.1.7.1 AddVariableValueAction

AddVariableValueAction is a write variable action for adding values to a variable. It is mapped to an ODRA assignment statement (if isReplaceAll=true) or to an ODRA *create*

temporal statement (if isReplaceAll=false).

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
AddVariableValueAction (isReplaceAll=true)	*x*.variable.name **:=** *x*.value **;**
AddVariableValueAction (isReplaceAll=false)	**create temporal** *x*.variable.name **(***x*.value**) ;**

8.3.1.7.2 ClearVariableAction

ClearVariableAction is a variable action that removes all values of a variable. It is mapped to the ODRA delete statement.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
ClearVariableAction	**delete** *x*.variable.name **;**

8.3.1.7.3 RemoveVariableValueAction

RemoveVariableValueAction is a write variable action that removes values from variables.

It is mapped to the ODRA delete statement.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
RemoveVariableValueAction	**delete** *x*.variable.name **[***x*.removeAt**];**

8.3.1.7.4 VariableAction

VariableAction is an abstract class for actions that operate on a statically specified variable. Its mapping to ODRA code is defined by its concrete subclasses.

8.3.1.7.5 WriteVariableAction

WriteVariableAction is an abstract class for variable actions that change variable values. Its mapping to ODRA code is defined by its concrete subclasses.

8.3.1.8 Variable

Variables are elements for passing data between actions indirectly. A local variable stores values shared by the actions within a structured activity group but not accessible outside it. VIDE variables are mapped to ODRA variable declaration statements. If the type of a VIDE variable is a class, an ODRA variable of a reference type is created. Otherwise, a non-reference type is used.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
Variable (class type)	*x*.name **: ref** *x*.type.name**Class [** *x*.lower **..** *x*.upper **];**
Variable (non class type)	*x*.name **:** *Map2ODRA*(*x*.type) **[** *x*.lower **..** *x*.upper **];**

8.4 Activities

8.4.1 Mapping

8.4.1.1.1 Activity

An activity is the specification of parameterised behaviour as the coordinated sequencing of subordinate units whose individual elements are actions. The mapping of an Activity to

ODRA is the sequential execution of owned nodes.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
Activity	**{** *Map2ODRA**(*x*.node, **;**) **}**

8.4.1.1.2 ActivityEdge

ActivityEdge is an abstract class for the connections along which tokens flow between activity nodes. Its mapping to ODRA is defined by its concrete subclasses.

8.4.1.1.3 ActivityGroup

An activity group is an abstract class for defining sets of nodes and edges in an activity. Its mapping to ODRA code is defined by its concrete subclasses.

8.4.1.1.4 ActivityNode

An activity node is an abstract class for points in the flow of an activity connected by edges. Its mapping to ODRA code is defined by its concrete subclasses.

8.4.1.1.5 ActivityParameterNode

An activity parameter node is an object node for inputs and outputs to activities. It is just mapped to the result of mapping its parameter.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
ActivityParameterNode	*Map2ODRA*(*x*.parameter)

8.4.1.1.6 Behavior

Behavior is a specification of how its context classifier changes state over time. In VIDE, the mapping is available for one concrete subclass of Behavior – namely, Activity.

8.4.1.1.7 Clause

A clause is an element that represents a single branch of a conditional construct, including a test and a body section. It is mapped to the ODRA conditional statement.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
Clause	**if (** *Map2ODRA*(*x*.test) **)** *Map2ODRA*(*x*.body) **;**

8.4.1.1.8 ConditionalNode

A conditional node is a structured activity node that represents an exclusive choice among some number of alternatives. It is mapped to ODRA as a cascade of if-else statements obtained by appropriate concatenation of mappings of ConditionalNode's Clauses.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
ConditionalNode	*Map2ODRA**(*x*.clause, **else**)

8.4.1.1.9 ControlFlow

A control flow is an edge that starts an activity node after the previous one is finished. It is mapped to ODRA composition of commands (in fact it means juxtaposition).

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
ControlFlow	*Map2ODRA*(*x*.incoming) *Map2ODRA*(*x*.outgoing)

8.4.1.1.10 ControlNode

A control node is an abstract activity node that coordinates flows in an activity. In VIDE it is used only for the ForkNode. Its mapping to ODRA is defined by its concrete subclasses.

8.4.1.1.11 ExceptionHandler

An exception handler is an element that specifies a body to execute in case the specified exception occurs during the execution of the protected node. ExceptionHandler is mapped to an ODRA try-catch statement.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
ExceptionHandler	**try** **{** *Map2ODRA*(*x*.protectedNode) **}** **catch (***Map2ODRA*(*x*.exceptionInput) **:** *Map2ODRA*(*x*.exceptionType)**)** **{** *Map2ODRA*(*x*.handlerNode) **}**

8.4.1.1.12 ExecutableNode

An executable node is an abstract class for activity nodes that may be executed. Its mapping ODRA code is defined by its concrete subclasses.

8.4.1.1.13 ExpansionNode

An expansion node is an object node used to indicate a flow across the boundary of an expansion region. ExpansionNode inherits its mapping to ODRA from its generalization.

8.4.1.1.14 ExpansionRegion

An expansion region is a structured activity region that executes multiple times corresponding to elements of an input collection. It is mapped to **foreach** statement. The attribute *bodyPart* is inherited from its generalization StructuredActivityNode.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
ExpansionRegion	**foreach (** *Map2ODRA*(*x*.inputElement) **) do {** *Map2ODRA*(*x*.bodyPart) **}**

8.4.1.1.15 ForkNode

A fork node is a control node that splits a flow into multiple concurrent flows. Since ODRA does not support parallel execution, the ForkNodes are mapped the same way as ControlFlow, i.e. to sequential execution of nodes.

8.4.1.1.16 LoopNode

A loop node is a structured activity node that represents a loop with setup, test, and body sections. LoopNode is mapped to an ODRA while-do (or do-while) statement preceded by the mapping of the setupPart.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
LoopNode (isTestedFirst=true)	*Map2ODRA*(*x*.setupPart) **;** **while (** *Map2ODRA*(*x*.test) **) do {** *Map2ODRA*(*x*.bodyPart)

	}
LoopNode (isTestedFirst=false)	*Map2ODRA*(*x*.setupPart) **;** **do {** *Map2ODRA*(*x*.bodyPart) **while (** *Map2ODRA*(*x*.test) **) ;**

8.4.1.1.17 ObjectFlow

An object flow is an activity edge that can have objects or data passing along it. It is not mapped directly to ODRA. It mapping amounts to assigning the integration of its inputPin with its outputPin, i.e. the mapping of its inputPin becomes the mapping of its outputPin.

8.4.1.1.18 ObjectNode

An object node is an abstract activity node that is part of defining object flow in an activity. The mapping of an ObjectNode to ODRA is inherited from a proper subclass of the TypedElement class.

8.4.1.1.19 SequenceNode

A sequence node is a structured activity node that executes its actions in order. The mapping of a SequenceNode to ODRA is just the sequential execution of owned subnodes.

Mapped node *x*	**Mapping result *Map2ODRA*(*x*)**
SequenceNode	*Map2ODRA**(*x*.executableNode, **;**)

8.4.1.1.20 StructuredActivityNode

A structured activity node is an executable activity node that may have subordinate nodes. Its mapping to ODRA is defined by its concrete subclasses.

8.5 Expressions

8.5.1 Mapping

8.5.1.1.1 CallExp

A CallExp is an expression that refers to a feature (operation, property) or to a predefined iterator for collections. Its result value is the evaluation of the corresponding feature. This is an abstract metaclass. Its mapping to ODRA is given by it concrete subclasses.

8.5.1.1.2 FeatureCallExp

A FeatureCallExp expression is an expression that refers to a feature that is defined for a Classifier in the UML model to which this expression is attached. Its mapping to ODRA is given by it concrete subclasses.

8.5.1.1.3 IfExp

An IfExp results in one of two alternative expressions depending on the evaluated value of a condition. It is mapped to the ODRA if-then-else-expression.

Mapped node *x*	**Mapping result *Map2ODRA*(*x*)**

IfExp	**if** *Map2ODRA*(*x*.Condition) **then**
	Map2ODRA(*x*.ThenExpression)
	else

Map2ODRA(*x*.elseExpression)

8.5.1.1.4 IterateExp

An IterateExp is an expression that evaluates its body expression for each element of a collection. This is the construct with the most complex mapping to ODRA. The ODRA operation *leaves by* has to be used:

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
IterateExp	**(***Map2ODRA*(*x*.baseExp) **groupas C).** **(** **(1 as counter,** *Map2ODRA*(*x*.setup) **groupas** *Map2ODRA***(***x*.result**))** **leaves by** **(** **((C[counter] as** *Map2ODRA*(*x*.iterator)**).(counter + 1 as counter,** *Map2ODRA*(*x*.body) **groupas** *Map2ODRA*(*x*.result)**))** **where counter <= count(C)** **)** **).***Map2ODRA*(*x*.result)

8.5.1.1.5 IteratorExp

An IteratorExp is an expression that evaluates its body expression for each element of a collection. It is mapped to a call to ODRA non-algebraic operator.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
IteratorExp	*Map2ODRA*(*x*.iterator) *Map2ODRA*(*x*.operator) *Map2ODRA*(*x*.body)

The mapping of OCL operators to ODRA operators is presented in the following table:

OCL operator	ODRA operator
->collect	. (dot)
->sortedBy	order by
->select	where
->exists	exists
->forAll	forall

Table 10: Mapping OCL iterator operations to ODRA SBQL

LiteralExp

A LiteralExp is an expression with no arguments producing a value. In general the result value is identical with the expression symbol. Literal expressions are mapped directly to ODRA. One exception is the string which is has to surrounded by double quotes in ODRA.

8.5.1.1.6 LoopExp

A LoopExp is an expression that represents a loop construct over a collection. It mapping to ODRA is defined by its concrete subclasses.

8.5.1.1.7 NavigationCallExp

A NavigationCallExp is a reference to a Property defined in a UML model. It mapped to a call to ODRA dot operator.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
NavigationCallExp	*Map2ODRA*(*x*.qualifier) **.** *x*.navigationSource.name

8.5.1.1.8 OclExpression

An OclExpression is an expression that can be evaluated in a given environment. OclExpression is the abstract superclass of all other expressions in the metamodel. It mapping to ODRA is given by its concrete subclasses.

8.5.1.1.9 OclVariable

Variables are typed elements for passing data in expressions. Its mapping to ODRA is just the call to its name.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
OclVariable	*x*.name

8.5.1.1.10 OpaqueExpression

An opaque expression is an uninterpreted textual statement that denotes a (possibly empty) set of values when evaluated in a context. Its mapping to ODRA is just its interpretation.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
OpaqueExpression	*x*.body

8.5.1.1.11 OperationCallExp

An OperationCallExp refers to an operation performed on build in OCL types that is mostly operator calls such as: +, -,*, /, <, >, =, <>, <=, >=, not, xor, and, or, unary -. User defined operations are called by appropriate action. It is mapped to ODRA function call or operator call.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
OperationCallExp (referredOperation is one of +, -,*, /, <, >, =, <>, <=, >=, or, xor, and)	*Map2ODRA*(*x*.argument[1]) *x*.referredOperation.name *Map2ODRA*(*x*.argument[2])
OperationCallExp (referredOperation is one of not, unary-)	*x*.referredOperation.name *Map2ODRA*(*x*.argument[1])
OperationCallExp (all other possibilities)	*Map2ODRA*(*x*.argument[1]) **.** *x*.referredOperation.name **(** *Map2ODRA**(*x*.argument[2-*], **,**) **)**

8.5.1.1.12 PropertyCallExp

A PropertyCallExpression is a reference to an Attribute of a Classifier defined in a UML model. It is mapped to ODRA dot operator.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
PropertyCallExpression	*Map2ODRA*(*x*.qualifier) **.** *x*.referredProperty.name

8.5.1.1.13 VariableExp

A VariableExp is an expression that consists of a reference to a variable. It is mapped to the variable name.

Mapped node *x*	Mapping result *Map2ODRA*(*x*)
VariableExp	*x*.referredVariable.name

8.5.1.1.14 ExpressionInOcl

An expression in OCL is an expression that is written in OCL. Because in the abstract syntax OclExpression is defined recursively, the top of the abstract syntax tree is represented by ExpressionInOcl, and it is defined to be a subclass of the ValueSpecification metaclass from the UML core, as shown in. The ExpressionInOcl is mapped the same way as its owned expression in OCL.

Mapped node *x*	**Mapping result *Map2ODRA*(*x*)**
ExpressionInOcl	*Map2ODRA*(*x*.bodyExpression)

8.6 VIDE Web services compilation rules for ODRA platform

In this section we describe compilation rules for Web service elements from a VIDE model. Additionally to generic (target platform independent) view on the compilation process we describe concrete compilation scenarios.
In Web Services profile subsection we describe Web Service related VIDE UML profile enhancements. Common compilation schema is subsection where we provide generic rules and best practices for Web services model compilation. We do not prescribe any particular approach used at target platforms for handling Web Services there. In the last subsection – Compilation Scenarios – we go into ins and outs of Web services compilation for ODRA platform.

8.6.1 Web Services profile

Web Service related classes can be marked inside the model with **«ConsumedService»** and **«PublishedService»** stereotypes.

«ConsumedService»
Designates that class will represent a proxy to remote Web Service conforming to certain WSDL contract. Its operations are associated to remote Web method calls of given Web Service. Exact shape of this stereotype will be specified based on the design decisions in model compilers development in VIDE.

Generalizations
Class

Attributes

URL:String[1]	Address of Web Service contract definition
port:String[1]	name of Web Service port to use
service:String[1]	name of Web Service to use

«PublishedService»
Tells system that class should be exposed as Web Service endpoint.

Generalizations
Class

Attributes
url:String[1] points to URL where Web Service should be installed
Namespace:String[1] - defaults to (filtered) containing package global name

«PublishedOperation»

Marks those operations, which should be available as Web methods of that endpoint.

Generalizations
Operation

Additionally the following naming conventions are used:

WSDL	VIDE
Publishing and consuming	
(encoded) target namespace	containing package names
Port type	class name
operations names	operations names
Publishing only	
Service name	Name of class suffixed with "Service"
Port name	Name of class suffixed with "Port"

Table 11: VIDE-WSDL naming conventions

8.6.2 Generic Web Services compilation schema notes

Web services are represented as regular VIDE model elements marked with certain stereotypes. However because of their remote behaviour, they need to be treated in a special way during compilation.
For example a consumed service is visible in editor as normal class (and set of associated types generated from WSDL) and hence can be called from any other package. However, compiler needs to be aware of that fact and compile it using dedicated procedure. All calls to such remote proxy can still be compiled in standard way. Possible problem here is to maintain tight control over the way consumed service is realized inside a target platform.
Importing service to model means generating a static proxy stub packaged with all necessary types. Same procedure is usually done on target platforms level. This is sufficient if system creation starts from PSM and there is no already collected web service information from PIM level. However in our case, where such data already exists, it should not be dismissed (i.e. by deciphering again all information from WSDL contract). Doing so affirms that all dependent (on service proxy) model elements will have their calls working correctly. Recreating proxy from scratch can lead to inconsistencies between what VIDE user sees and what is being executed (hence errors would be less descriptive and debugging more problematic).

> We did not encountered this problem in our compilation scenarios, and hence do not prescribe following this more laborious mapping path here to prevent them. However, we want to make developers aware of possible implications of proxy regeneration. We can imagine that for certain use cases this will be a sufficient solution.

For published services similar discussion need to be made. In that case fortunately there is no need to manage tight control over Web service element because no code generation occurs. However what may be crucial to provide is to have exactly the same WSDL contract of service being exposed for each target platform. To achieve such effect, WSDL (at least some part of it) should be generated before compiler gets started. Instead of bare model, compiler will be feed in additionally with such (partial) contract. This requires usage of contract first

approach toolset to create service stub. Such skeleton can then be filled with target platform code.
In such approach compiler authors need to manage tight control over generated stub. This is important since class being exposed can be used also locally. Building service as a black-boxing wrapper on such local class is reasonable solution to follow.

> This feature should be considered optional. Because of its high implementation complexity and no direct requirement for this in neither of the two currently supported target platforms, it will not be implemented in VIDE prototype. In future it may be realized as additional Web Services VIDE component common to all target compilers and resisting between PIM and PSM layers.

8.6.3 The model compiler to ODRA

8.6.3.1 Services

In order to include Web Services definition and usage from the level of VIDE models we extend the metamodel with two class stereotypes. Web services are compiled in similar way to regular classes. However there are some exceptions from the standard compilation routine. The exact compilation routines for consumed and published services are described in the following subsections.

8.6.3.1.1 ConsumedService

To achieve tight control over compiling, the consumed services proxy is not regenerated using `add module … as proxy` ODRA DDL command. Instead of that regular compilation takes place (this also applies to containing package and associated types). Thanks to that no special handling for compilation of remote methods calls is necessary. Finally compiled class is promoted to constitute remote proxy using dedicated DDL ODRA command.

Mapped node *x*	**Mapping result *Map2ODRA*(*x*)**
Class marked with «ConsumedService» stereotype	*(after x.package compiled code)* cm x.package.name; *(port and service are taken from associated Web Service options file section)* promote x.nameClass to proxy on "self.getValue(self.getAppliedStereotypes()-> select(name='ConsumedService')->asSequence()->at(1), 'URL')" with (port="self.getValue(self.getAppliedStereotypes()-> select(name='ConsumedService')->asSequence()->at(1), 'port')", service=" self.getValue(self.getAppliedStereotypes()-> select(name='ConsumedService')->asSequence()->at(1), 'service')") cm ..

8.6.3.1.1.1 Example

Let's consider the following VIDE model `consuming-example.uml` taken from related WP5 document chapter.

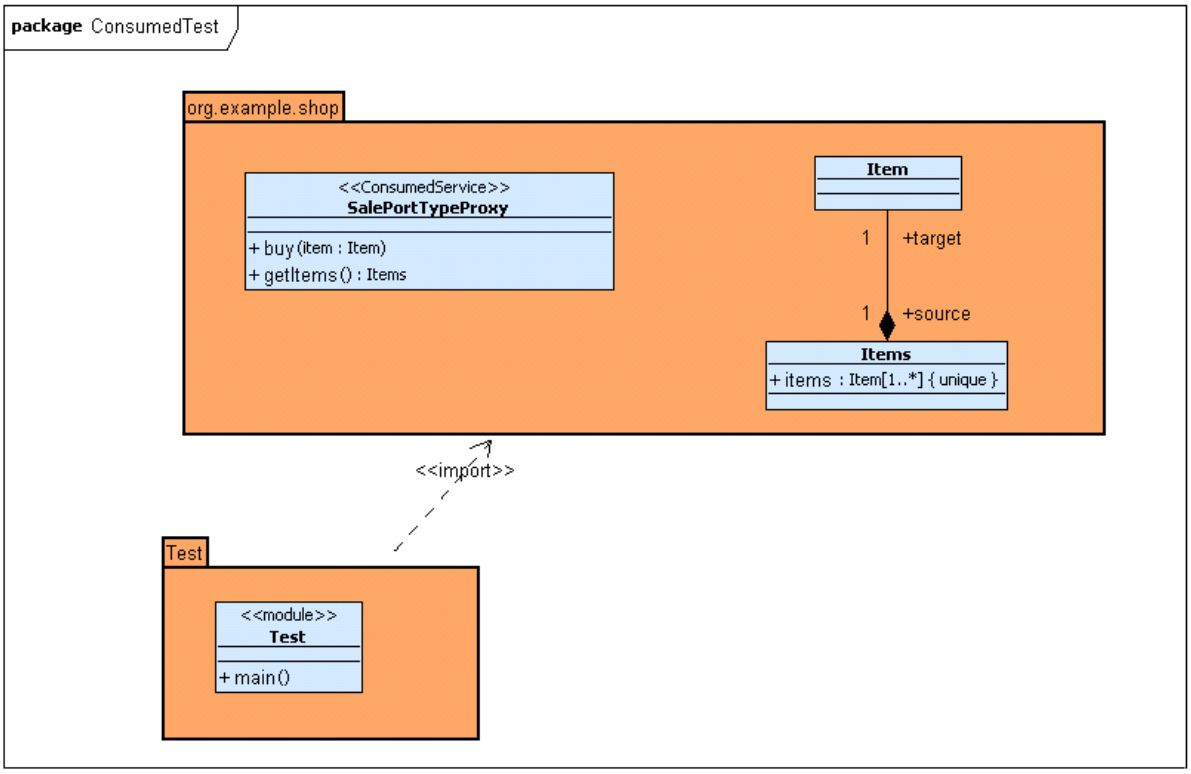


Figure 35 : Example for consumed service mapping into ODRA

Main procedure from the above diagram has the following body:

```
context Test::Test.main body
{
  let serviceProxy : SalePortTypeProxy = create { };
  if (serviceProxy.getItems()->size() > 0) {
    let toBuy : Item = serviceProxy.getItems()->first();
    if (toBuy.getPrice() < 100) {
      serviceProxy.buy(toBuy);
    }
  }
}
```

The result of compilation procedure described above will be:

```
add module org_shop_example {
  class SalePortTypeProxyClass {
    buy(item:Item) { }
    getItems():Items { }
  }
  class Item {
    // attributes
  }
  class Items {
    items:Item[1..*];
  }
}

cd org_example_shop
promote ShopSalesPortTypeProxyClass to proxy on
    "http://localhost:8080/Shop?wsdl" with (
  port="ShopSoap11Port",
  service="ShopService"
);
cd ..
```

```
add module Test {
  import org_shop_example;
  main() {
    salePortTypeProxy:SalePortTypeProxyClass;
    if (count(salePortTypeProxy.getItems()) > 0) {
      if (salePortTypeProxy[0].getPrice() < 100) {
        salePortTypeProxy.buy(toBuy);
      }
    }
  }
}
```

8.6.3.1.2 Publishing

Since ODRA does not support the *contract first* approach – the simple approach will be used for publishing. Since exposed components are regular (constrained) classes, their compilation will be handled by standard mapping routine. Thanks to that no special handling is necessary for compilation of local methods calls.
ODRA endpoint is not created on original class but on its wrapper. The wrapper contains only methods marked with PublishedService stereotype and relays real execution to the underlying class.
Finally dedicated ODRA DDL command is used to expose Web Service. ODRA supports only the wrapped document/literal service invocation style – if different one is requested, compilation error is reported.

Mapped node *x*	**Mapping result *Map2ODRA*(*x*)**

Class marked with «PublishedService» stereotype	class *x*.nameClass extends *Map2ODRA**(*x*.superClass, ,) {
	instance *x*.name {
	*Map2ODRA**(*x*.ownedAttribute, ;) ;
	}
	*Map2ODRA**(*x*.ownedOperation)
	}
	class x.nameWrapperClass extends x.nameClass {
	instance x.nameWrapper {
	internal:x.nameClass;
	}
	for each x.ownedOperation marked with «PublishedOperation» stereotype
	x.ownedOperation.name(
	*Map2ODRA**(*x*. ownedOperation.ownedParameter, ;))
	: *Map2ODRA* (*x*.ownedOperation.ownerParamater[*i*].type) {
	internal.x.ownedOperation.nameOperation(
	Map2ODRA*(x.ownedOperation.ownedParameter.name, ,));
	}
	}
	(after x.package compiled code)
	cm x.package.name;
	(url, port and service are taken from associated Web Service options file section)
	add endpoint x.nameEndpoint on x.nameWrapperClass with (
	state=STARTED,

	path="/relativePart(self.getValue(self.getAppliedStereotypes()->select(name='PublishedService')->asSequence()->at(1), 'URL'))", portType="x.name", port="x.namePort", service="x.nameService", ns="self.getValue(self.getAppliedStereotypes()->select(name='PublishedService')->asSequence()->at(1), 'namespace')") cm ..

Class marked with «Module» and «PublishedService» stereotype	class *x*.nameWrapperClass extends *Map2ODRA**(*x*.superClass, ,) {
	instance *x*.name {
	}
	}
	for each x.ownedOperation marked with «PublishedOperation» stereotype
	x.ownedOperation.nameOperation(
	*Map2ODRA**(*x*. ownedOperation.ownedParameter, ;))
	: *Map2ODRA* (*x*.ownedOperation.ownerParamater[*i*].type) {
	x.ownedOperation.name(
	Map2ODRA*(x.ownedOperation.ownedParameter.name, ,));
	}
	}
	(after x.package compiled code)
	(same as for the above case)

8.6.3.1.2.1 Example

Let's consider the following VIDE model `publishing-example.uml` taken from related WP5 document chapter.

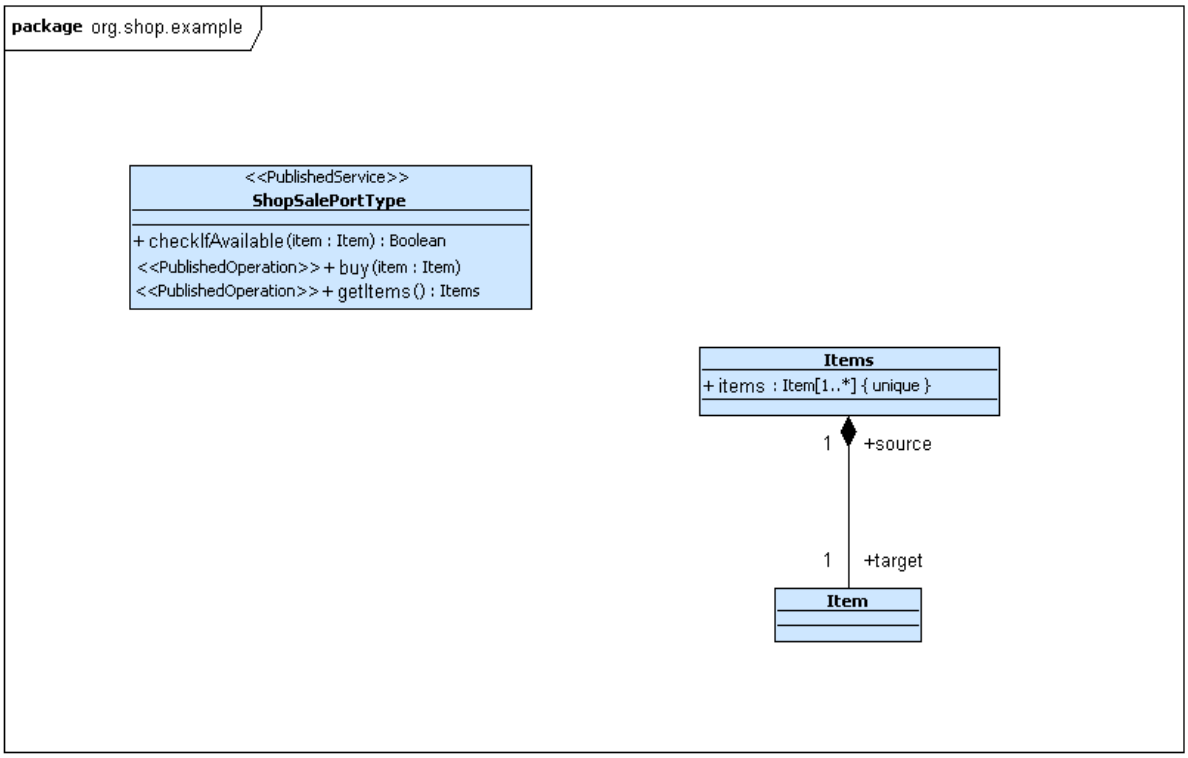


Figure 36 : Example for published service mapping into ODRA

The result of compilation to ODRA using routine described above will be:

```
add module org_shop_example {
  items:Items;
  class ShopSalePortTypeClass {
    instance ShopSalePortType : { }
    checkIfAvailable(item:Item):boolean { ... }
    buy(item:Item) { ... }
    getItems():Item[0..*] { ... }
  }
  class Item {
    // attributes
  }
  class Items {
    items:Item[1..*];
  }
  class ShopSalePortTypeWrapperClass extends ShopSalePortTypeClass {
    instance ShopSalePortTypeWrapper {
      internal:ShopSalePortTypeClass;
    }

    buy(item:Item) { internal.buy(item); }
    getItems():Item[0..*] { return internal.getItems(); }
  }
}

cm org_shop_example
add endpoint ShopSalePortTypeEndpoint on ShopSalesPortTypeWrapperClass with
(STATE=STARTED, path="/Shop", portType="ShopSalePortType",
port="ShopSalePortTypePort" service="ShopSalePortTypeService"
ns="example.shop.org");
```

9 Transformation frameworks

Following the MDA approach, models created by VIDE editors are supposed to be platform independent, whereas some model compiler generates a platform specific model or even code towards a specific target platform. In this context, we evaluated several MDA tools with respect to their usability as underlying framework for the VIDE model compiler to Java.

In the following, we will present some evaluation criteria and especially VIDE-specific ones. Then we give a brief overview and evaluation of the most promising MDA tools with respect to the Java model compiler.

9.1 Evaluation Criteria

9.1.1 Requirements defined by VIDE specification

Generally, the underlying generator tool should support the development of a model compiler as described in [VIDE2007a] in the specification of work package 6:

- The model compiler should *"exemplify the mapping of Action Semantics representation into common application server platforms, thus allowing to verify VIDE completeness and flexibility in the development targeted onto typical commercial software platform"*.
- The model compiler should enable the development of the prototype (to be developed in work package 9).

Consequently, compatibility to other VIDE modules is crucial, especially with regard to the development of an integrated prototype. To ensure this, the technology chosen for the model compiler should be selected carefully and the requirements collected during work package 1 should be taken. These requirements are presented in the following.

9.1.1.1 Integration with the Eclipse

As pointed out in [VIDE2007a] and [VIDE2007b], the VIDE system including the VIDE prototype will be developed using the Eclipse framework, because Eclipse is considered as being a successful, widely adopted Open Source project. Therefore, the MDA tool must provide integration in the eclipse framework.

9.1.1.2 Compatibility with EMF

The VIDE partners decided to use EMF as VIDE's modelling framework for the PIM modelling with UML. For model storage and to be interoperable with existing UML modelling tools, the Ecore-based UML2 implementation of MDT is used, as it nicely integrates the OCL metamodel. UML2 export is now supported by many tools.

Therefore, the MDA tool used by the model compiler must accept UML2 models (serialized in XMI) as input.

9.1.1.3 Supporting Multiple Target Platforms

Following the MDA approach, VIDE should support code generation for various target environments.

Therefore, the MDA tool should not be limited to a specific target language and should be extensible in regard to new target platforms.

9.1.1.4 Open Source

In [VIDE2007a], VIDE is specified to be an open and interoperable platform, that will be compliant and build upon standards (UML/XMI) and successful open source platforms for tool interoperability. The MDA tool used by the model compiler to Java should also be Open Source.

9.1.1.5 Using XPand as Model-to-Text transformation language

In the context of WP 1 [VIDE2007a] §7.4.4, several Model-to-text transformation standards were compared. In particular we compared the Velocity template Language, and XPand. XPand has several advantages over Velocity as it is simple and easy-to-learn (less than 10 commands), natively support MDSD as it takes real models as input, strongly typed and thus supports syntax checking while editing.

Requirement Tool-5 in D1.1 states that XPand should be used for Model-to-text transformation in VIDE.

9.1.2 Other Criteria

Additionally, there are some more common tool features to be mentioned, which are not specific to the VIDE project. As the, they can indicate the maturity and quality of the MDA tool, they should be taken into consideration, too.

9.1.2.1 Industrial Adopted Tool

The tool should be proven in real-world industrial projects.

9.1.2.2 Tool Documentation and Support

Comprehensive, up-to-date tool documentation should be available. The tool should be sufficiently maintained or further developed. Support via forum or e-mail should be available.

9.2 Tool Evaluation

9.2.1 Overview

The general discussion about the possibilities and opportunities which the Model Driven Software Development potentially offers has led to a wide choice of MDA tools, which all claim to support Model Driven Architecture.

Far more than 60 tools can be investigated incorporating at least one of the major aspects of MDA:

- UML-based modelling
- Transformation between the application overall design models and the models that are specific to the underlying computing architecture
- Generation of code in a specific language

To be suitable for the specific task of WP6 in the VIDE context, a model compiler should support especially the third aspect, code generation.

On the other hand, the first aspect, UML-based modelling, is not in the specified scope, as the model compiler should integrate with a VIDE editor based on EMF (chapter 2.1.2). Therefore, there is no need for an own graphical UML modelling facility. Consequently, these MDA modelling suites are considered inappropriate.

Filtering the remaining selection for widely-used Open Source projects reveals that there are only two candidates left: AndroMDA and openArchitectureWare. In the following sections, these two will be shortly introduced and then we evaluate whether they are conform with the remaining criteria specified in Section 2 and thus suitable as underlying MDA tool for the VIDE model compiler.

9.2.2 AndroMDA

AndroMDA is described as extensible generator framework, following the MDA paradigm. AndroMDA takes a UML model from a CASE-tool as input and generates classes and deployable components for all kinds of platforms.

AndroMDA comes with a big bundle of ready-to-use metamodels and templates (cartridges), making it easy to get started. There is a cartridge from UML to Java but it supports only the structural part of UML (i.e., no action support).

The current stable version is AndroMDA 3.2.

9.2.2.1 Integration with Eclipse

There is currently no stable Eclipse IDE integration. It is pronounced that this will change in the near future, as an integration project called Android is on the edge of being released.

9.2.2.2 Compatibility with EMF

Since the current version of AndroMDA (3.2), EMF UML2 compatible XMI files are supported.

9.2.2.3 Support of Multiple Target Platforms

AndroMDA can generate (textual) code for any target platform. It comes with a bundle of ready-to-use metamodels and templates, making it easy to get started with simple projects for various platforms, e.g.: Struts, JSF, Spring, Hibernate, EJB und jBPM. If these cartridges will not fit the current requirements a new cartridge can be developed

9.2.2.4 Open Source

AndroMDA is Open Source.

9.2.2.5 Integration of XPand as transformation language

AndroMDA uses the open source-Framework Velocity from Apache Software Foundation as template engine. An integration of XPand is not provided.

9.2.2.6 Industrial Adopted Tool

AndroMDA is widely-used as several success stories can be found (e.g. used by Lufthansa Systems).

9.2.2.7 Tool Documentation and Support

AndroMDA provides some tool documentation but the documentation seems outdated at some points. Support can be obtained at the forum, which seems to be frequently read by a large community.

9.2.3 OpenArchitectureWare

OpenArchitectureWare (oAW) is a modular MDA/MDD generator framework implemented in Java. It supports parsing of arbitrary models, and a language family to check and transform models as well as generate code based on them. Supporting editors are based on the Eclipse platform.

At the core, there is a workflow engine allowing the definition of generator/transformation workflows. A number of pre-built workflow components can be used for reading and instantiating models, checking them for constraint violations, transforming them into other models and then finally, for generating code.

Current stable version is oAW 4.2.

9.2.3.1 Integration with Eclipse

oAW is a subproject of the Eclipse Modeling project. Therefore, it is smoothly integrated in Eclipse.

9.2.3.2 Compatibility to EMF

OAW has strong support for EMF UML2-based or Ecore-based models but can also work with other models, too (e.g. XML or simple JavaBeans).

9.2.3.3 Support of Multiple Target Platforms

Any (textual) artifact can be generated using the XPand generator. Therefore, multiple target platforms are supported. Only a few cartridges for standard platforms are ready-to-use available. The developers describe oAW as “tool for building tools”; their goal is not to develop generators but rather to provide the underlying framework, enabling the users to build their own generator.

9.2.3.4 Open Source

oAW is Open Source.

9.2.3.5 Integration of XPand as transformation language

XPand is an integral part of oAW.

9.2.3.6 Industrial Adopted Tool

oAW is widely-used, several success stories can be found on the tool homepage.

9.2.3.7 Tool Documentation and Support

Parallel to the release of oAW4.2 the documentation was completely revised. A direct contact to the oAW developer team is possible via the English or German forum.

9.3 Evaluation Results

Comparing AndroMDA and oAW according to the chosen criteria, it is obvious that oAW is regarded as the favourite:

	AndroMDA	oAW
Integration with Eclipse	-	+
Compatibility to EMF	Not possible in version 3.1	+
Support of multiple target platform	+	Just a framework
Open Source	+	+
Integration of XPand as transformation language	-	+
Industrial Adopted Tool	+	+
Tool Documentation and Support	not up-to-date, tutorials missing	+

Table 12: AndrMDA vs. oAW comparison table

AndroMDA fails at the Criterion of Eclipse and XPand integration.

Although it seems that AndroMDA can gather some points with its ready-to-use cartridges for all kinds of target platforms, this start-up advantage is withdrawn in the context of the VIDE model compiler as the peculiarities of VIDE (UML actions and OCL expressions) would result in the need for newly developed cartridges. The procedure of developing a generator from scratch is certainly better supported by oAW, as the developer profits from the highly advanced set of editors integrated in the oAW framework.

Additionally, the oAW framework with its modular-structured architecture and comprehensive set of languages (e.g., Xtend, Check, etc) and the respective user-friendly editors promises more flexibility with regard to the integration in the overall VIDE framework.

10 UML Metamodel evolution propositions

As presented Figure 37, in UML, an Activity is not owned by an Operation but by the Class to which the Operation belongs. This model is difficult to understand and offer very little reuse of the Activity because they depend heavily on the parameters of the Operation.

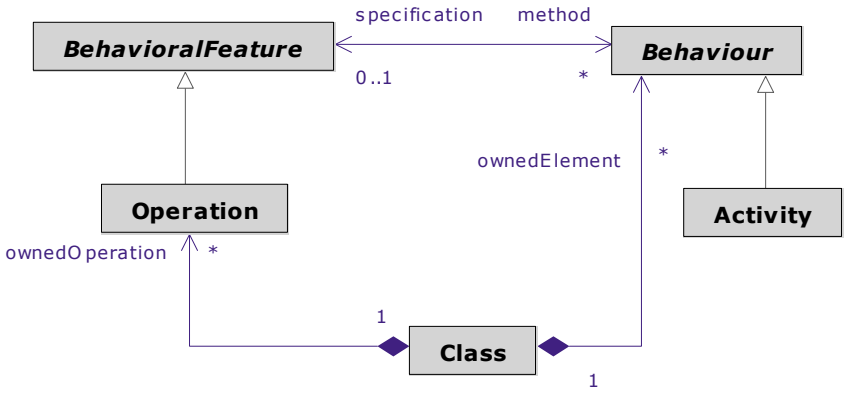


Figure 37 : Activity ownership in UML Metamodel

Therefore, we propose the more understandable and manageable metamodel presented Figure 38, where an Activity that describes an Operation is owned by this Operation. Note that the composition between Class and Behaviour still exist to manage Activity defined at the class level (it is not shown here to ease the understanding of the modification). The BehavioralFeature class, of little help, is removed.

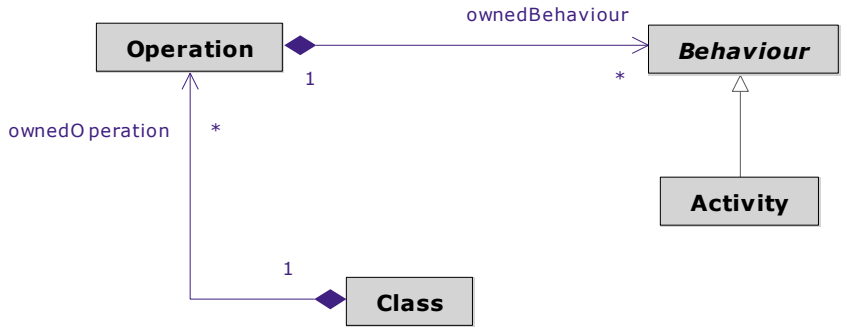


Figure 38 : Activity ownership proposition

A number of minor issues have been identified in the area of the Activities unit and its integration with the remaining part of UML. This part of specification is relatively new and is seldom being implemented.

The most significant problems are related with attempting to use OCL as a general purpose query language for UML (though the specification explicitly mentions this as one of the OCL purposes). Namely, the following issues may need resolving in the further revisions of UML and OCL specifications.

- OCL cannot access UML's Variable element. There are no appropriate expressions in the standard. It can read value from a Property of a Class or a Parameter of an Operation but cannot read values from a Variable defined in a StructuredActivityNode.
- Unspecified conversion of OCL types to UML types. Although there is a conversion specified from UML types to OCL types, there is no explicit definition of the opposite conversion. It is then formally impossible to consume OCL expression results in UML actions and other UML constructs
- Consuming of OCL collection types in UML actions. There is an important problem of correct and common interpretation of collection types. In UML a collection is represented by multiple values. OCL defined dedicated collection types, which are containers for stored values. When OCL expression is accessing UML multiple value, it is converted to appropriate OCL collection instance. On the other side also OCL expression (or query) may return multiple values, which are packed in a collection type. However, from UML's point of view, OCL collection is just a single value (of a collection, say Bag type). There is no reverse mapping from OCL collections to UML multiple value variables. Because of that, standard UML cannot treat OCL collections properly and cannot handle them for example in Expansion Regions. Such a conversion also cannot be done implicitly when consuming OCL results in ValueSpecificationAction. UML specification says that type of *ValueSpecification* in this action must be the same as the type of result in the *OutputPin* ([UML2007] p.302).
- Finally, it should be noted that introducing the truly seamless support for OCL expressions for UML behaviour would make a number of actions redundant: (*ReadStructuralFeatureAction*, *ReadSelfAction*, *ReadExtentAction*, *ReadLinkAction* etc.) thus contributing to the simplification of the overall metamodel.

Moreover, the way OCL expressions can be embedded into UML behaviour is rather redundant for the purpose of realizing expressions inside method bodies. The wrapping provided by *ExpressionInOcl* class instance, constructing the expression's environment (e.g. the *self* variable etc.) each time, even for the most trivial expressions results in a very large number of objects inside the model repository.

11 Conclusion

The aim of the “Model Compilers” work package is to specify the mapping of VIDE language to several languages and execution platforms.

This study has been done for two different target languages and execution platforms, the Java/J2EE and ODRA SBQL languages. The first is a well known general purpose object oriented programming language while the other belongs to a new brand of object oriented programming language that integrates database query to its core and designated for rapid development of business intensive application.

The mapping to Java has been done in three steps:

1. Mapping to plain Java without any other consideration than to find semantically accurate and efficient translation schemas
2. Mapping to JPA to allow VIDE program to interact transparently with databases. This mapping integrate not only model navigation as it is common in object oriented paradigm, but also the support for limited but useful queries based on JPQL.
3. Mapping to web services, using the annotations defined in JAX-WS standard API. The mapping to web services is bidirectional: the compiler can generate code to produce publish web services as well as generating code to call access externally defined web services.

ODRA mapping provided a quite straightforward way of achieving executable form of the VIDE models. For this reason, the ODRA engine was chosen to be adapted and provided with the editor facilities in order to allow model execution at development time, directly from the VIDE editor.

Through the point of view of the mapping, we have shown the validity and the completeness of the VIDE metamodel and pinpoint some simplifications of the underlying UML metamodel to enhance its usability.

Beyond the specification of the mapping for Java, the available tools to implement that mapping have been studied with respect of the requirements that come from WP1 work and that prescribe the use of a Xpand template base transformation tool that is integrated with Eclipse platform. OpenArchitectureWare has been chosen for that purpose.

The implementation of these mappings and the use of the resulting tool will undoubtedly make appear enhancements and optimizations of the presented work, as well as the opportunity for the support of other platforms like .NET.

12 Glossary

CIM – Computation Independent Model. A high level, abstract model of a given problem domain, focusing on requirements and environment of the system.

DBMS – Data Base Management System. It is a suite of programs which support the management and accessing of large structured sets of persistent and shared data. DBMSs are widely used in business applications. A current DBMS is an extremely complex set of software programs that controls the organization, storage and retrieval of data (or objects) in a database. It also supports many features related to data management such as buffer management, authorization of users, granting privileges for users, database schema and sub-schema management, security, integrity, consistency, privacy, client-server architecture, query optimization and processing, concurrent access to shared data (transaction processing), data abstractions such as views, triggers and stored procedures, various interoperability facilities, geographic distribution of resources, and others. A DBMS frequently equipped with additional facilities such as Web interfaces, integrated programming languages, graphical user interfaces, data warehouses, report and form generators, multimedia management (graphics, voice, video), and others. Currently the most popular DBMSs are based on the relational model and SQL as a query/programming language. Other datamodels, in particular, object-oriented and XML-oriented, are also considered as a basis for the DBMS construction.

JPA – Java Persistence API. The Java Persistence API provides a POJO persistence model for object-relational mapping. The Java Persistence API was developed by the EJB 3.0 software expert group as part of JSR 220, but its use is not limited to EJB software components. It can also be used directly by web applications and application clients, and even outside the Java EE platform, for example, in Java SE applications.

JPQL – Java Persistence Query Language. The Java Persistence query language defines queries for entities and their persistent state. The query language allows to write portable queries that work regardless of the underlying data store.

MDA – Model Driven Architecture. It is an initiative promoted by OMG, assuming the central role of models (in particular – platform independent models (PIM)) in the software development process. Support for automated model transformations plays an important role for productive application of this vision.

Metamodel extension – modification of the metamodel of existing modelling language to provide it with additional or modified features needed for particular area of application. Less intrusive ways of extension assume defining annotations or stereotypes, while the more intrusive ones assume addition or modification of the metamodel classes.

MOF – Meta Object Facility. It is an OMG specification that defines a meta-metamodel for specification of various modelling languages in its terms. It is intended to provide a common foundation for those languages to support the development of common frameworks for models construction and transformations which is essential for realising the MDA postulates.

oAW – OpenArchitectureWare. openArchitectureWare (oAW) is a modular MDA/MDD generator framework implemented in Java. It supports parsing of arbitrary models, and

a language family to check and transform models as well as generate code based on them.

OCL – Object Constraint Language. An OMG specified language being developed in association with the UML. Its main purpose is supporting the modelling and metamodelling by specifying precise constraints over their instances. Moreover, one of the potential applications of OCL is providing query language functionality.

OMG – Object Management Group. Is a consortium established to for setting standards in the areas of object-oriented distributed systems and software modelling. Significant OMG initiatives or standard specifications include MDA, UML and CORBA.

PIM – Platform Independent Model. Is a model of software specified in the way that avoids dependency on particular technological platform.

Query language – The term is used in two contexts: (1) A language allowing the users for quick, ad hoc retrieval in large data resources. There are many such languages, frequently based on natural language processing, graphical tools or some forms. (2) A language that is used as an application programming interface to access databases, e.g. SQL, OQL, OCL, SBQL, JPQL, XQuery, etc. It is usually assumed that a query language should possess the following properties: high abstraction level, no involving physical details of data, non-procedurality (declarativity), macroscopic processing (many-data-at-a-time), simplicity and naturalness for programmers, machine efficiency for very large databases due to query optimization, universality (covering majority of useful user requests), domain independence, interpreted rather than compiled. Query languages, notably SQL, are considered the main factor of the spectacular success of the relational database technology. Queries are also building blocks for programming and database abstractions such as views, triggers, stored procedures and constraints.

Service Oriented Architecture – an approach to software architecture aimed at reuse and loose coupling between applications, postulating modularisation in the form of services, which are usually by degree of magnitude more course grained than in earlier approaches to reuse and modularisation. To adjust to the changing requirements, the services are intended to be easily composable in the process of *orchestration*.

UML – Unified Modeling Language. The language is designed for broad area of application in terms of various problem domains and levels of precision. Recent versions provide extended set of modelling constructs in order to make it possible to realize with UML the concept of executable modelling. Current published version is UML 2.1.1 [OMG2007b].

UML Action Semantics – a part of UML language included into its metamodel in version 1.4 of the specification, in order to provide it with the means of specifying behavioural details like reads, updates, collection processing and control flow. In subsequent versions of UML the respective features have been redesigned and included in Actions and Activities units.

UML Actions – a UML unit grouping elementary constructs of behavioural specifications. Those include particularly object and link reads, updates, removals, variable reads and updates, performing calls and sending signals.

UML Activities – a UML unit serving for behavioural modelling. Its notions allow describing the sequence and conditions for executing lower level behaviours. The concepts of control flow and object flow are emphasized. Apart from them, the package

StructuredActivities introduces the elements for structured style modelling, which allow for relatively straightforward definition of constructs found in typical programming languages.

UML Classes – the foundational unit of the UML static structure modelling notions. It defines the elements and notation for class models and the model decomposition mechanism using the notion of Package.

UML Components unit – a UML unit depending on UML Classes, designed for specification of logical and physical components. Particularly, its notions allow for decomposition and definition of interfaces among replaceable software units.

Web services – middleware technology assuming reuse and integration of application functionality by providing it with a form of course grained services available through standard protocols and reusing the WWW infrastructure.

13 References

[VIDE2007a] Annex I - “Description of Work” (amendment). 2007-04-24. VIDE Consortium 2007.

[VIDE2007b] Deliverable number D.1.1: Standards, Technological and Research-Base for the VIDE Project, Project Evaluation Criteria and User Requirements Definition. VIDE Consortium 2007.

[VIDE2007c] Deliverable number D.2.1: VIDE language definition. VIDE Consortium 2007.

[SAPAS] SAP Netweaver Application Server
http://www.sap.com/platform/netweaver/components/applicationserver/index.epx

[SAPJPA] Getting Started with Java Persistence API and SAP JPA 1.0
https://www.sdn.sap.com/irj/sdn/go/portal/prtroot/docs/library/uuid/40ff8a3d-065a-2910-2f84-a222e03d1f43

[ADHK2008] R. Adamus, M. Daczkowski, P. Habela, K. Kaczmarski, T. Kowalski, M. Lentner, T. Pieciukiewicz, K. Stencel, K. Subieta, M. Trzaska, T. Wardziak, J. Wiślicki: Overview of the Project ODRA. 1st International Conference on Object-Oriented Databases, Berlin 13-14 March 2008.

[BEAJPA] Documentation on JPA
http://edocs.bea.com/kodo/docs41/full/html/ejb3_overview.html

[JAX] JAX-WS Annotations
https://jax-ws.dev.java.net/jax-ws-ea3/docs/annotations.html

[OMG2007] OMG: Unified Modeling Language Specification (Superstructure and Infrastructure) Version 2.1.2. November 2007
http://www.omg.org/spec/UML/2.1.2

Disclaimer of SAP AG[1]

[1] Applies to Sections 6.2, 6.3, 7.2, and 9