TECHNISCHE UNIVERSITÄT BERGAKADEMIE FREIBERG

The University of Resources. Since 1765.

Automatic Generation of Software Applications - A Platform-based MDA Approach

By the Faculty of Mathematik und Informatik
of the Technische Universität Bergakademie Freiberg

approved

Thesis

to attain the academic degree of

Doktor-Ingenieur
(Dr.-Ing.)

submitted by M.Sc. Dong Liang

born on the June 21, 1981 in Dalian, China

Assessors: Prof. Dr.-Ing. habil. Bernd Steinbach
Prof. Dr.-Ing. Wilfried Schubert

Date of the award: Freiberg, 30th April 2014

Versicherung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Hilfe eines Promotionsberaters habe ich nicht in Anspruch genommen. Weitere Personen haben von mir keine geldwerten Leistungen für Arbeiten erhalten, die nicht als solche kenntlich gemacht worden sind. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

30. April 2014 M.Sc. Dong Liang

Declaration

I hereby declare that I completed this work without any improper help from a third party and without using any aids other than those cited. All ideas derived directly or indirectly from other sources are identified as such.

I did not seek the help of a professional doctorate-consultant. Only those persons identified as having done so received any financial payment from me for any work done for me. This thesis has not previously been published in the same or a similar form in Germany or abroad.

30th April 2014 M.Sc. Dong Liang

Acknowledgments

I would like to express my heartfelt gratitude to my first supervisor, Prof. Dr. Bernd Steinbach. Without your trust, encouragement and effort to provide me financial support during my PHD study, I could never start this thesis. Without your patience to spend long hours with me discussing the organization of this thesis and many technical details involved in this thesis, this work would have never been completed. Additionally, I would also like to give my sincere appreciation to my second supervisor, Prof. Dr. Wilfried Schubert, who agreed very kindly to read my thesis and gave me so many valuable and constructive suggestions.

I am particularly grateful to Mr. K.-A. Bebber, one of the leaders of the Human Resources Services division in the Bayer Business Services GmbH. In long-term cooperation with our institute, he and his company supported the research of our group to solve many problems of practical applications and provided me the scholarship during my PHD-study.

Special thanks are given to Matthias Werner and Michael von Wenckstern, who helped me so much to improve my dissertation with Latex; Prof. Dr. Jasper and Matthias Lenk, who discussed with me about fresh ideas; Carlos Döring, who extended MOCCA in his master thesis.

Last but not least, many thanks are also to my parents, parents in law as well as my loving wife, Jun. Without your patience, encouragement and emotional support, I would have never been so brave and confident to face challenges.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

List of Abbreviations

ABAP	Advanced Business Application Programming
AL	Action Language
API	Application Programming Interface
AS	Application Server
AST	Abstract Syntax Tree
BNF	Backus-Naur Form
CASE	Computer-Aided Software Engineering
CRUD	Create, Read, Update and Delete
CST	Concrete Syntax Tree
DM	Design Model
DOM	Document Object Model
DPM	Design Platform Model
DSL	Domain Specific Language
DVDL	Device and Visualization Definition Language
EIS	Enterprise Information System
EMF	Eclipse Modeling Framework
GUI	Graphical User Interface
IDE	Integrated Development Environment
ILPM	Implementation Language Platform Model
JEE	Java Enterprise Edition
JNDI	Java Naming and Directory Interface
JPA	Java Persistence API
JSE	Java Standard Edition
MAL	MOCCA Action Language

MDA	Model Driven Architecture
MDE	Model Driven Engineering
MES	Manufacturing Execution System
MOCCA	Model Compiler for generating Complete Applications
MOF	Meta Object Facility
O/R Mapping	Object Relational Mapping
OCL	Object Constraint Language
OMG	Object Management Group
OOA	Object-Oriented Analysis
OOD	Object-Orient Design
OOPL	Object-Oriented Programming Language
PIM	Platform Independent Model
QVT	Query, View and Transformation
TM	Target Model
TPM	Target Platform Model
UML	Unified Modeling Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
WPF	Windows Presentation Foundation
XML	eXtensible Markup Language
XOCL	eXecutable OCL
XSD	XML Schema Definition

Glossary

ABAP

Advanced Business Application Programming is a programming language that was developed by SAP for developing commercial applications in the SAP environment. [KK07]

BNF

Backus-Naur Form is a notation used to describe context free grammars. The notation breaks down the grammar into a series of rules, which are used to describe how the programming languages tokens form different logical units.[Coo13]

DM

Design Model is a platform independent UML model representing the main application logic. It contains model elements defined by application modelers or provided in DPM. The XOCL expressions specifying behaviors are also stored in design model. Design model is the most important input for the MOCCA model compiler.

DPM

Design Platform Model is a UML model that comprises a set of ready-to-use data types, their relationships, and constraints as well as important meta-data, which serve as the foundation to build a platform independent design model.

DVDL

The Device and Visualization Definition Language is a proprietary script language, which is developed by the company Apromace data systems GmbH. DVDL is used to develop Apromace's own MES product. [Apr13]

EJB

Written in Java programming language, Enterprise Java Beans are server-side components that encapsulate business logic and take care of transactions and security.

JPA

The Java Persistence API provides Java developers with an object/relational mapping facility for managing relational data in Java applications.

LALR(1)-Parser and LALR(1)-Grammar

An LALR-parser is a variant of LR Parser, which parses the input from Left to right, and constructs a Rightmost derivation. Formally the LALR parser generally refers to the LALR(1)-parser with the "1" denoting one-token lookahead. LALR parsers are bottom-up parsers. The class of grammars, from which an LALR(1)-parser can be constructed is called LALR(1)-grammars.

LL(k)-Parser and LL(k)-Grammar

An LL-parser is a top-down parser for a subset of the context-free grammars. It parses the input from Left to right, and constructs a Leftmost derivation of the sentence. The class of grammars which are parsable in this way is known as the LL-grammars. An LL-parser is called an LL(k) parser if it uses k tokens of lookahead when parsing a sentence. If such a parser exists for a certain grammar and it can parse sentences of this grammar without backtracking then it is called an LL(k)-grammar. [Wik13]

Mapping Configuration File

The Mapping Configuration File is an XML file, which defines the elementary mapping rules between DPM and TPM model elements.

MOCCA Configuration File

The MOCCA Configuration File, which is also called MOCCA Project File, is used to define the global configuration parameters for the compiler environment.

TM

Target Model is a platform specific UML model that realizes a platform independent DM with the resources provided on a specific target platform. A TM is produced by a model mapper for a concrete target platform.

TPM

Target Platform Model is a UML model that describes a concrete technical platform, on which the implementation language and its API are provided. For example, there are TPM for the Java Standard Edition platform and TPM for the Java Enterprise Edition, etc.

1. Introduction

1.1. Motivation

Great progress in the software development was made in the last decade. It is evident from the fact that it is feasible to build much more complex and larger systems. One important reason is the adoption of the object-oriented software development paradigm [Bal05] [Bar05] [LR09], which has profound impact on different activities within software development process. For implementation Object-Oriented Programming Languages (OOPL) like C++, Java and C# have been widely used and proven their values in different kinds of projects, ranging from sophisticated distributed enterprise applications, over personal desktop applications to the recently emerging smart device based applications. Methodologies for analysis and design have been unified by adopting object-orientation. As described in [Bal05] Object-Oriented Analysis (OOA) is done to establish a system model concentrating only on problem domain to reflect user requirements. Such an OOA model will be refined into an Object-Oriented Design (OOD) model by taking into account technical constraints on a concrete implementation platform. To build OOA and OOD models, the Unified Modeling Language (UML) [OMG10b] and the Object Constraint Language (OCL) [OMG10a] have become the de facto standard. As a visual modeling language, the UML provides many graphical notations to represent popular object-oriented concepts like *class*, *interface*, *inheritance* and so on, whereas the OCL is used to write compact expressions specifying constraints and complex queries.

Even though the methodologies applied in analysis, design and implementation phase have been unified, transition from an OOA-model to the associated OOD-model and then to the implementation is not straightforward, because this process has to add plenty of details to an OOA-model to reflect the technical characteristics of a selected implementation platform. To explore which kinds of details to be added, a typical application can be dissected into components dealing with [LB10]:

- Core concern
- Cross-cutting concern
- Plumbing

The core concern comprises the *business logic* that an application must implement. The cross-cutting concern contains secondary operations like security assertions, transactional boundaries, concurrency policies, which are necessary for core concern running correctly and efficiently. Plumbing are infrastructures like data transmission over network or methods of database connection. It is obvious that business logic are unique and domain specific whereas cross-cutting concern and plumbing are platform specific and once implemented well on a platform can be reused for different applications targeted on this platform.

With these characteristics in mind, the process of refining an OOA-model into an OOD-model and finally transforming the OOD-model into an implementation can be considered as a closed interval $[OOA - model, OOPL - implementation]$. On the one hand, an OOA-model should be compact and concentrate on business logic only. On the other hand, the OOPL selected as implementation platform should be rich enough, that means solutions for important cross-cutting concern and plumbing should have been provided and their implementation details are hidden from application developers. Thanks to the reusability and information encapsulation principles coming with object-orientation the most modern OOPLs and their frameworks, e.g., the Java Enterprise Edition (JEE) [Ora11b], the .Net framework [Tro07] as well as the SAP Application Server ABAP [KK07], have been powerful enough to provide such auxiliary services for sophisticated application development. By using such an implementation platform it helps to smooth the transformation from an OOA-model into its implementation. If the transformation can be taken over by a tool automatically, then the software development process will be moved from low-level implementation to high-level modeling. That is where the Model Driven Architecture (MDA) [OMG03] comes into play.

MDA suggests that software developers model their software products in a platform independent way. Such a software model is called Platform Independent Model (PIM). Then an MDA-tool is used to transform the PIM into one or more Platform Specific Models (PSM). The PSMs have involved detailed information for implementation. Hence, code generation from a PSM is straightforward. In order to realize model transformation, two different strategies are feasible. One of them defines the model transformation process in a high-level specification language. The Query, View and Transformation language (QVT) [OMG11b] supported by OMG is the de facto standard for this strategy. The QVT allows to define a transformation in an imperative approach. Then, a QVT compiler generates an implementation (e.g., in Java) of this transformation specified in the QVT source file as a model compiler, which is dedicated to this transformation. The other strategy is to develop the model compiler itself as an all-purpose model transformation framework as well as to provide all necessary information about the underlying target platform the PSM based on, in the form of Target Platform Models (TPM). To prove the second strategy, the model compiler MOCCA (Model Compiler for reConfigurable Architecture) [Frö07] was developed by Dr. Fröhlich in our institute. The set of applications, which were initially targeted to MOCCA, are application-specific accelerators for logic optimization problems, neural networks, and multimedia applications. These applications expose mixed control- and data-flow and can be modeled using UML class diagrams together with MOCCA Action Language (MAL), which is a Java-like action language (AL) designed for MOCCA.

In this thesis MOCCA is enhanced to deal with more general purpose applications like GUI-based desktop applications and distributed enterprise applications based on the *three-layer-architecture* [Fow+02]. Due to these significant enhancements, a new meaning, *MOdel Compiler for generating Complete Application* is assigned to MOCCA. In this thesis, the acronym MOCCA refers always to this new meaning. If the early version of MOCCA developed by Dr. Fröhlich is meant, enough context will be given to distinguish one from the other.

1.2. Contributions and Constraints

As mentioned in the previous section, the primary objectives of this thesis are enhancements in the modeling methodology for creating compact PIM as well as the upgrade of the MOCCA compiler for model transformation in the context of MDA. To accomplish these objectives, major challenging issues coming with MDA are re-studied carefully and summarized as follows:

1. Creating a PIM compactly, completely and correctly in UML is difficult. Because UML is a very powerful and universal modeling language that can be used to model different aspects of a software from many different perspectives in different degrees of detail.

2. The transformation from a PIM into different PSMs is complicated and very difficult to maintain. Even if a manageable manner of creating PIMs could be found, the technical details on target platforms, on which PSMs base, can vary from time to time.

It is easy to find that the first issue involves three aspects:

1. compactness means simplification in comparison with a potential concrete PSM,
2. completeness requires that both structure and behavior of a PIM must be modeled completely and
3. correctness means that the established PIM is both syntactically and semantically correct to fulfill the requirements of an application.

The primary objectives of this thesis are to find novel solutions to cope with the challenging issues above based on the research achievements of Dr. Fröhlich [Frö07]. As an early attempt to explore the MDA methodology, Dr. Fröhlich created the first complete and consistent approach to map and synthesize the applications for run-time reconfigurable computer architectures from their UML-models and realized this approach into his MOCCA model compiler. For the platform independent modeling, this early version of MOCCA was equipped with a Design Platform Model (DPM) providing fundamental modeling concepts like primitive data types and their operations, basic IO-facility, etc., which built the foundation for creating platform independent design models. To specify behaviors in the design models, a Java-like MOCCA Action Language was developed, which supported elementary imperative instructions to describe computation-intensive logic in detail. For model transformation, novel algorithms for mapping, estimation, synthesis, and run-time management of applications of run-time reconfigurable architectures from UML design models were developed and implemented in the corresponding compiler components. Hence, the final achievements of the early version of MOCCA can be highlighted as:

- modeling application-specific accelerators for logic optimization problems, neural networks, etc., based on software-hardware co-design in a platform independent way,
- transforming such design models into complete and runnable applications targeted to run-time reconfigurable computer architecture with VHDL script for hardware description and C++ source code for implementing software modules, respectively.

During my research of enhancing MOCCA to generate software applications with more general purpose, several shortcomings of the early version of MOCCA are discovered and can be summarized as follows:

- The fundamental data types are not based on any well-known standard and lack support for other common modeling issues in a clean manner, such as modeling collections, modeling GUI system of an software application, involving design pattern in the application model, etc.
- The Java-like MOCCA Action Language is considered too elementary to fulfill the requirement of compact behavioral modeling.
- The UML metamodel implementation in the early version of MOCCA was developed before the UML 2 standard was published, Hence, it is not identical to the one used by our own CASE tool UML 2 Designer [SBL04]. The latter conforms to the latest UML 2 standard much better. Hence, extra effort is required to translate models between the both tools.
- The elementary mapping rules defining mappings between the model elements in design platform model (DPM) and target platform model (TPM) are embedded in the TPM directly by means of *dependency* between two types, that makes a TPM have always to know about the DPM.

The major contributions of this thesis with respect to both efficient modeling and model transformation can be highlighted as follows:

- The OCL library types and their predefined operations are adopted into the MOCCA DPM for modeling primitive and collection types.
- For compact behavioral modeling, a new action language based on OCL are developed by extending OCL with necessary syntactical constructs for the missing action semantics. The resulted language is named *eXecutable OCL* (XOCL), which is very OCL-like but a full-fledged action language.
- A platform independent GUI toolkit is developed by abstracting a limited set of common GUI elements from diverse target platforms.
- Proposal is given to use the XOCL-expressions to involve the layout information into a GUI model in a platform independent manner.
- To model event handling, a concise XOCL-event-expression is developed to connect an event to its handling method in a platform independent way.
- A DPM profile containing diverse domain specific and design pattern based stereotypes is developed to create design models with high-level semantics.
- MOCCA is enhanced to adopt the metamodel implementation of the UML 2 Designer as its internal model representation.
- To support XOCL, new front-end components for parsing and validating XOCL-expressions are developed.
- An XML-based *mapping configuration file* is developed to define the elementary mappings between DPM and TPM resources in a more flexible manner, which replaces the elementary mapping rules in the TPM of the early version of MOCCA.
- New model mappers for the Java Standard Edition platform [Ora12], the Java Enterprise Edition platform [Ora11a] and the SAP NetWeaver Application Server ABAP [Kel05] [KK07] [FK08] platform are developed and integrated into the MOCCA model compiler to enhance the ability of MOCCA for generating software application.

The MDA itself is a huge research topic. Hence, only restricted aspects of MDA can be studied in the scope of the presented thesis. Back to the challenging issues exposed at the beginning of this section, one of notable restrictions to MOCCA is that the correctness of a PIM can be checked very limited. However, this issue is not considered as a serious issue, because the final product of MOCCA is the implementation code of an application, which can be compiled and run directly. Another major restriction of MOCCA concerns dealing with deployment issues for an application. For example, deploying mission-critical enterprise applications onto different technical platforms involves usages of diverse artifacts like deployment descriptors of the JEE platform. The same philosophy is shared by deploying applications targeted on special hardware platforms, e.g., modern smart devices, etc. For certain highly proprietary platform like SAP NetWeaver Application Server ABAP, a deployment from outside the system is extremely difficult. Hence, modeling deployment for a software application in UML and generating the corresponding deployment plan for a possible target platform remains an interesting and challenging research topic.

1.3. Overview

The chapters of this thesis are organized top-down in the order of the typical development flow. Since each chapter builds on the issues and notations discussed in the previous chapters, the chapters ought to be read in order.

Chapter 2 - Theoretical Foundation and Related Work discusses the fundamentals and technological context of the presented work. A brief introduction is given to the Unified Modeling Language (UML), the Object Constraint Language (OCL) as well as the Model Driven Architecture (MDA) respectively. The presented thesis is completely based on these three topics. Moreover, the important related work, especially in terms of the present tools is reviewed.

Chapter 3 - The MOCCA Modeling Framework addresses both structural and behavioral modeling for creating compact and complete platform independent design model in detail. The complete structure of a design model is created in UML class diagram with the help of a well defined modeling library called design platform model, whereas behaviors are modeled using a full-fledged action language called executable OCL (XOCL).

Chapter 4 - The MOCCA Model Compiler presents the tool, which can be considered as a model compiler, to support the modeling technique developed in Chapter 3. The working principle of each important compiler component is discussed in detail. For back-end components like model mapper and code generator, which can be spawned for diverse target platforms but are based on common developing principle, either the Java Standard Edition or the Java Enterprise Edition is chosen as target platform to expose the developing principle, depending on the implementation details in discussion.

Chapter 5 - Experimental Results shows the practical applications of both modeling technique and the tool support.

Chapter 6 - Conclusions concludes this thesis and discusses import future directions of research in this field.

2. Theoretical Foundation and Related Work

2.1. UML - The de facto Standard Software Modeling Language

2.1.1. Overview of the UML Diagram Types

The UML is a non-proprietary language for the visual specification, design and documentation of complex software systems. UML is independent of specific domains, architectures, and methods. This language was introduced in 1997 in response to the demand for uniform and consistent notations for object-oriented software systems and processes. Since UML was released, there have been several revisions. The most important revision was the upgrade from UML 1.x to UML 2.0. The most recent UML 2.5 standard consists of 13 diagram types, which can be classified as shown in Figure 2.1.

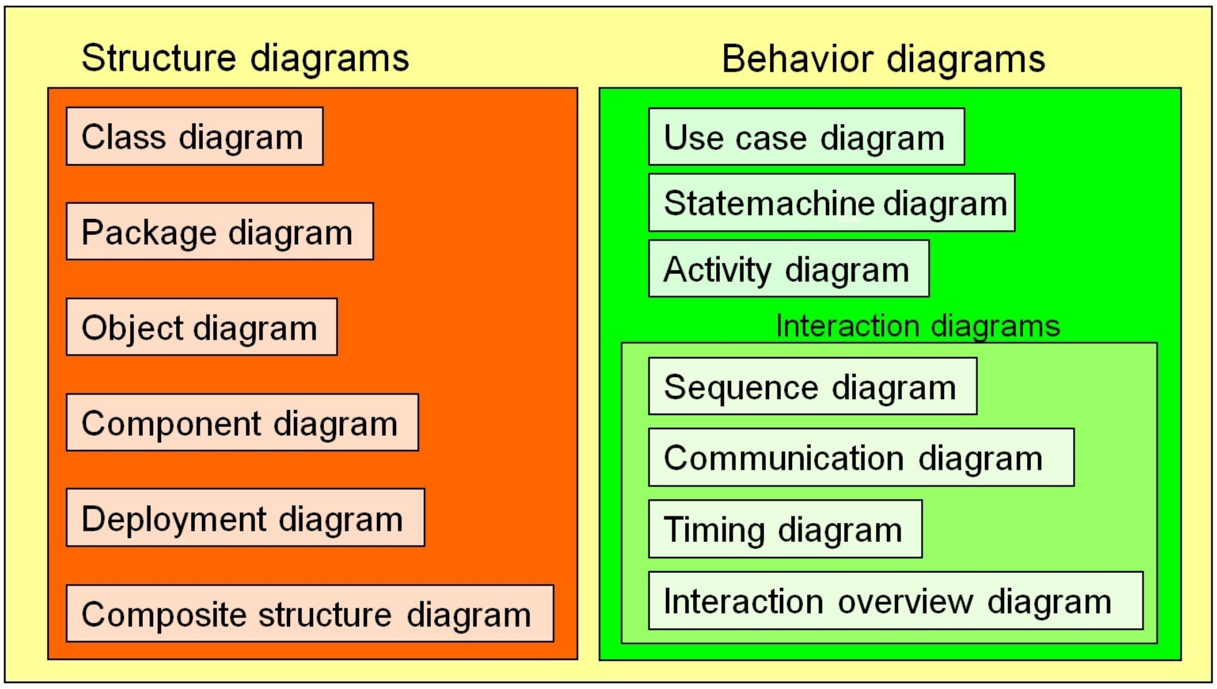


Fig. 2.1.: UML diagram types and their classification

Structure diagrams illustrate the static features of a model, they are used to capture the physical organization of the things in a system, whereas behavior diagrams describe how the elements modeled in the structure diagrams interact with each other and how they execute their capabilities. As explained in [Pen03] and [RQZ07], each of these diagram types finds its usage to model different aspects of a software system with various degrees of detail. According to our primary objective of generating complete source code for software application from UML model, only class diagram and package diagram, which can be merged into class diagram implicitly, are chosen to establish static model of a software system. The model elements supported by class diagram and package diagram cover almost all essential concepts coming with OO-technology.

Instead of using UML behavior diagrams, text-based XOCL is used as AL to create precise and compact behavior model. About this decision detailed discussion is found in Section 3.1.

2.1.2. The UML Metamodel

The OMG makes difference between four metamodel layers as shown in Figure 2.2:

- Instance-layer (M0) contains concrete data (objects) of a running system, which are instances of the model elements of the M1-layer. For example, the *student1* in Figure 2.2 is an instance of class *Student* whose attribute *studentID* is assigned with a concrete value.
- Model-layer (M1) represents the model itself of a software system. The model elements of this layer have instances on the M0-layer at runtime. The model is created in a modeling language, e.g., the UML, which is defined by its own metamodel on the M2-layer.
- Metamodel-layer (M2) contains elements defining the modeling language used on M1-layer. Instances of this layer are model elements on the model-layer. For example the UML class *Student* is an instance of the metaclass *Class* of this layer.
- Metametamodel-layer (M3) is in fact the Meta Object Facility (MOF) [OMG11a] according to the OMG terminology. The elements of this layer are used to define metamodel of a language. For example, the UML metaclass *Class* and *Property* are instances of the model element *Class* of this layer. To avoid an infinite metamodel layering, elements of M3 can be represented by the concepts on the same layer.

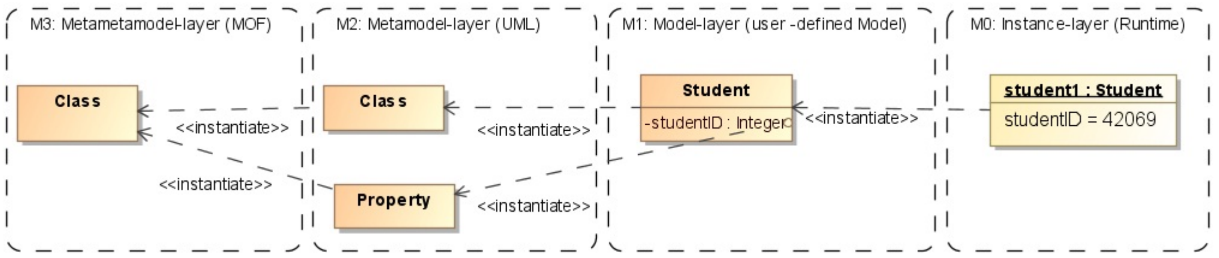


Fig. 2.2.: The OMG four layered metamodel architecture

Up to this point it is clear that metamodel of arbitrary modeling language used on M1-layer can be defined by using model elements provided in MOF. The UML 2 Metamodel is one of them and has been defined and documented in [OMG10b] by OMG. Figure 2.3 shows a snippet of the UML metamodel. The concrete metaclasses represent UML language constructs, which can be used directly to establish a model, whereas many other abstract metaclasses serve as structuring and reducing the entire metamodel hierarchy. The composition associations between metaclass *Class* and *Property*, *Operation* as well as *Classifier* define that a UML class can own arbitrary number of properties, operations, even nested classes.

Knowing about UML metamodel is fundamental to understand the internal representation of UML models in a CASE-Tool like *MagicDraw* or our own *UML 2 Designer* on the one hand. For CASE-Tools implemented in Java programming language, there is a Java implementation of the (part of) UML metamodel, whose instances represent the actual model being edited. On the other hand, the model transformation, which is the key topic of this dissertation, is done by manipulating the PIMs represented in the form of metaclass instances.

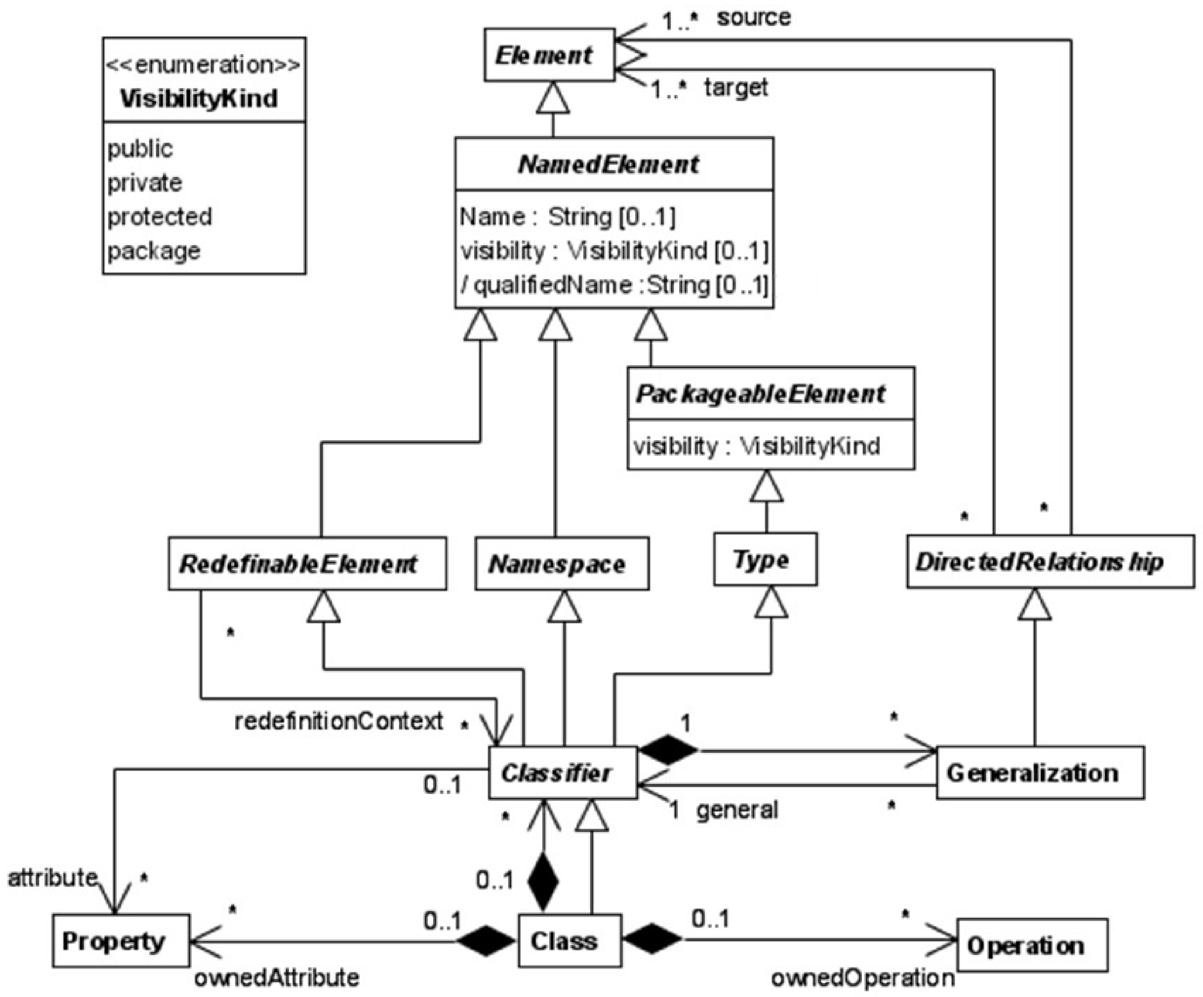


Fig. 2.3.: A snippet of UML metamodel

2.1.3. Profile - Lightweight Extension Mechanism of UML

With the knowledge of the previous section it should be easy to understand that UML can be extended by creating appropriate metaclasses and metarelationships. An extension in this way modifies or upgrades the UML metamodel itself, thus, can be considered as heavyweight. A lightweight extension of UML is the profile mechanism.

A *profile* defines limited extensions to a reference metamodel with purpose of adapting the metamodel to a specific platform or domain [OMG10b]. In essence a profile is an extension package containing stereotypes and constraints. A *stereotype* defines how an existing metaclass may be extended, and enables the use of platform or domain specific terminology or notation in place of or in addition to the ones used for the extended metaclass [OMG10b]. A stereotype can have properties, which are called *tagged values* in this context.

Figure 2.4 illustrates a simple example. Stereotype *«Application»* is defined in the profile *DPMProfile* by extending UML metaclass *Class*. The extension relationship means that an instance of stereotype *«Application»* can be applied to an instance of the metaclass *Class*. On the right hand side of Figure 2.4 the application of this stereotype is shown. By using a stereotype, additional information can be added onto a normal model element, which is especially significant for model transformation. For example, the stereotyped UML class *Payback* can be transformed into Java class annotated by *@Startup* for JEE platform or a normal Java class that must con-

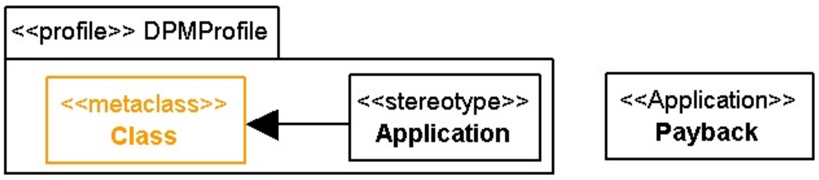


Fig. 2.4.: A simple stereotype definition and its application

tain a *public static void main(String[ ] args)* method for JSE platform. Because the stereotype *«Application»* defined in MOCCA DPM, which will be discussed in detail in Section 3.3.4, has the semantics marking a UML class as execution entrance. Thus, the platform oriented model mapper can interpret this stereotype appropriately.

2.1.4. The UML Action Semantics

Early versions of the UML did not prescribe a way of precisely defining behavior. In order to fill this gap, an action semantics specification has been incorporated into the specification since the version 1.5 was released. As a response to MDA, which suggests using models for more than just documentation or informal design sketching, the UML 2.0 provides a much more extensive and systematic coverage of semantics relative to earlier versions.

However, as explained in [Sel04], the UML specification does not cover the semantic aspects in a focused fashion. Instead, due to the idiosyncrasies of the format used for standards, the material is scattered throughout the documents, making it very difficult to develop a consistent global picture. In order to solve the problem above, a clean defined *semantics architecture* with high level view of the semantics definitions of standard UML and their relations was given in [Sel04] and has been involved in [OMG10b].

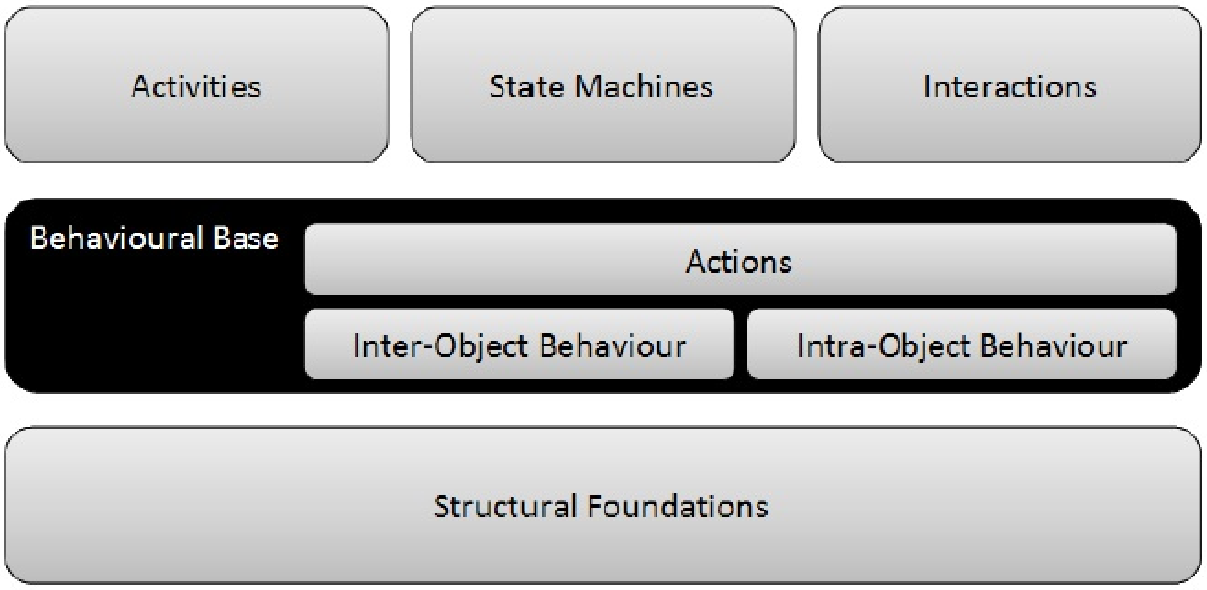


Fig. 2.5.: The UML semantics layers

Figure 2.5 [Bro+06] shows this semantics architecture based on the concepts discussed in [Sel04]. At the highest level of abstraction, it is possible to distinguish three distinct layers of semantics:

- Structural foundations reflect the premise that there is no disembodied behavior in UML - all behavior emanates from the actions of structural entities.
- Behavioral base provides the foundation for the semantic description of all higher level behavioral formalisms. This layer consists in fact three separate sub-areas arranged into two sub-layers. The inter-object behavior deals with how structural entities communicate with each other, whereas the intra-object behavior defines essentials for representing actions and for combining them. The actions sub-layer defines the semantics of individual actions and means by which actions are composed to form more complex behavioral specifications.
- The topmost layer in the semantics hierarchy defines the semantics of the behavioral formalism of UML: *activities*, *state machines* and *interactions*. In UML behavior diagram types are provided to support these high level behavioral semantics.

An action is the fundamental unit of behavior specification. An action takes a set of inputs and converts them into a set of outputs, though either or both sets may be empty. Actions are contained in behaviors, which provide their context. Behaviors provide constraints among actions to determine when they execute and what inputs they have. A primitive action like *TestIdentityAction* or *ReadVariableAction* carries out a computation or accesses object memory, but never both. This approach enables clean mappings to a physical model. It is usually necessary to combine a number of primitive actions to get an overall desired effect. This requires *flow control* organizing primitive actions to produce more complex behavior specification. Both the primitive actions and control flow elements defined in UML action semantics can be represented with different graphical notations in different concrete behavior diagrams according to the topmost layer of the semantics architecture.

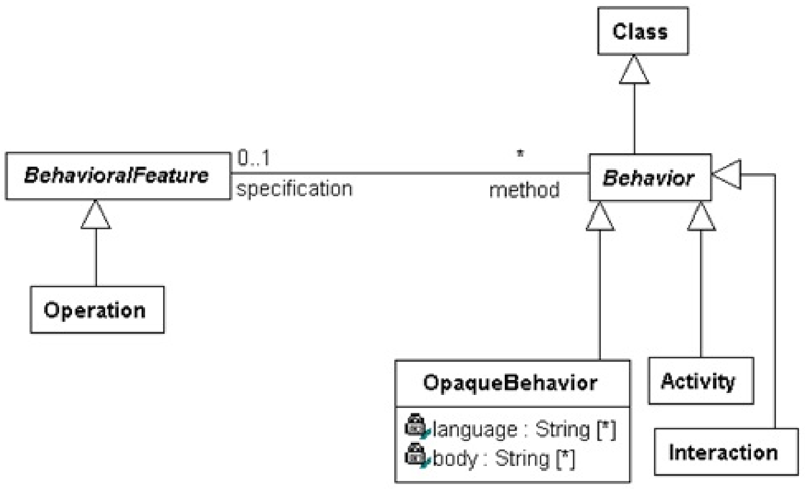


Fig. 2.6.: The relationship between structure and behavior in the UML metamodel

Another way to encompass the UML action semantics in a model is to define a surface action language. A particular AL could implement each semantic construct one-to-one, or it could define higher level, composite constructs to offer the modeler both power and convenience. For example, creating an object may involve initializing attribute values or creating objects for mandatory associations. The specification defines the create action to only create the object, and requires further actions to initialize attribute values and create objects for mandatory associations. A high-level AL could choose to define a creation operation with initialization as a single unit as a shorthand for several actions.

As mentioned in Section 1.2, one of the primary objectives of this work is to build a complete, MDA-ready software model covering both static structure and dynamic behavior. It is clear that static aspects involving *operation signature* can be modeled well in UML class diagram, whereas behavior can be modeled from different points of view in both ready-to-use behavior diagram types and text-based AL. The UML supports a unified way to combine structural modeling and behavioral modeling smoothly. Figure 2.6 shows this combination on the metamodel (M2) layer. The actual combination is done by the association between both abstract metaclasses *BehavioralFeature* and *Behavior*. That means, an operation can be further specified by one of the concrete behaviors. If the activity diagram is used, the operation instance links to an activity object. If an textual AL is used, the linked behavior instance has the type *OpaqueBehavior*, whose meta-attributes indicate the AL in use and the code itself.

2.2. Domain Specific Language

2.2.1. Abstract and Concrete Syntax

The basic idea of a domain specific language (DSL) is a computer language that is targeted to a particular kind of problem, rather than a general purpose language (Java, C# or UML) that is aimed at any kind of software problem [Fow08]. The most common DSLs in use today are textual, e.g., SQL [WC05] and OCL. But graphical DSL [Lab13] exists, too.

Like general purpose language, a DSL is defined by both its *abstract syntax* and *concrete syntax*. The former defines the significant semantic aspects of a DSL, which usually correspond to the domain, in which this DSL is used, whereas the latter depicts what a DSL looks like.

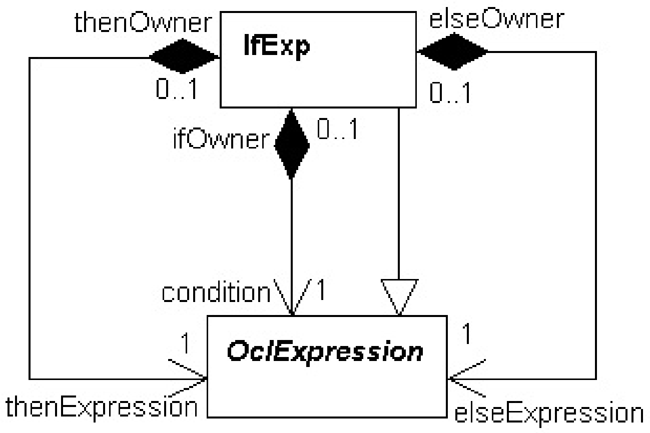


Fig. 2.7.: Abstract syntax metamodel for OCL-if expression

Figure 2.7 shows a snippet of the official OCL abstract syntax [OMG10a] in the form of the metamodel. Thus, it is aware that the UML metamodel described in Section 2.1.2 defines the abstract syntax of the UML language. The abstract syntax shown in the metamodel above defines that each OCL *if*-expression contains one OCL *expression* as condition, one as *then*-expression and another one as *else*-expression. Because the metaclass *IfExp* is an *OclExpression*, the OCL *if*-expression can be used within both of the *then*-branch and *else*-branch to compose a nested style.

The concrete syntax of a textual DSL is defined by *grammar rules* written in Backus-Naur Form (BNF) notations. Listing 2.1 shows the grammar rule deriving the OCL if-expression.

```
<IfExpCS> ::=  'if' <OCLExpCS> 'then' <OCLExpCS>  'else' <OCLExpCS> 'endif'
```

Listing 2.1: Grammar rules deriving OCL if–expression

2.2.2. The Object Constraint Language

The Object Constraint Language is a DSL used to describe expressions on UML models. Expressions written in OCL rely on the types (i.e., the classes, interfaces and so on) that are defined in the UML diagrams. These expressions typically specify *invariants* that must hold for system being modeled or *queries* over objects described in a model. One of the most important peculiarities of OCL is that it is a pure *declarative language* without side effects, i.e., the state of the corresponding executing system cannot be altered when OCL expressions are evaluated.

Two reasons make OCL possible to become a compact and complete AL used in this dissertation to model behaviors: OCL is a formal language with rigorous but simple syntax; the OCL predefined data types and their operations are powerful and easy to use.

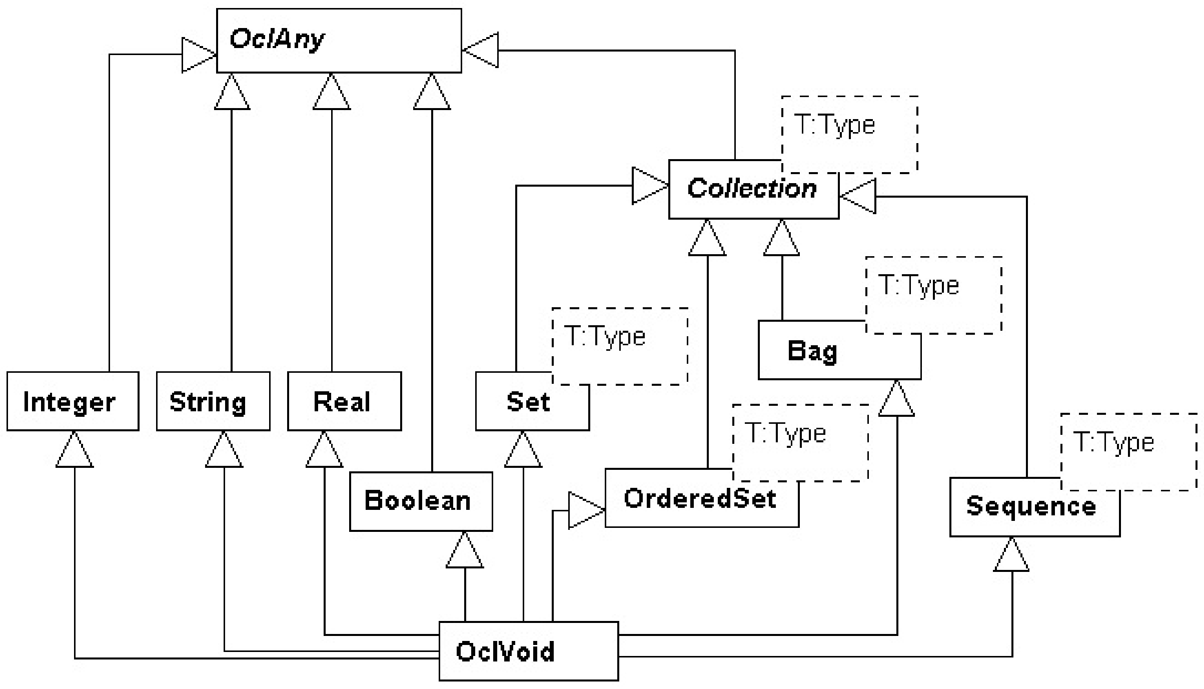


Fig. 2.8.: The predefined data types in OCL

Figure 2.8 shows the most important OCL predefined data types, which can be divided into three groups:

- *OclAny* and *OclVoid* are auxiliary types. *OclAny* defines a number of operations useful for every type of OCL instance. Therefore, the type *OclAny* is considered to be the super-type of all types in the model. All predefined types and all user-defined types inherit its features. Because OCL is a strongly typed language, if some expression is evaluated as undefined, the *OclVoid* can be used to indicate this result.

- *Integer*, *Real*, *String* and *Boolean* are primitive data types, which find usages in almost all the software systems. Primitive data types define necessary arithmetic operations and string manipulation operations.

- Within OCL there are five collection types. The *Set*, *OrderedSet*, *Bag* and *Sequence* are concrete types in expressions. *Collection* is the abstract super-type of other four and is used to define the operations common to all collection types.

In OCL expressions collection types can be used explicitly to define local variables or obtained implicitly by specifying multiplicity of operation parameters, properties of class, and even navigating from one class to another connected by association. Because in object-oriented system, one-to-one associations are rare, most associations define a relationship between one object and a collection of other objects. In Figure 2.9 the association navigating from *ProgramPartner* to *Service* defines a property named *deliveredServices* in *ProgramPartner* as a collection of *Service* objects. Which OCL collection type is exactly in use can be configured as depicted in Table 2.1.

Tab. 2.1.: Taxonomy of the OCL collection types

unique	ordered	collection type
X		Set
X	X	OrderedSet
		Bag
	X	Sequence

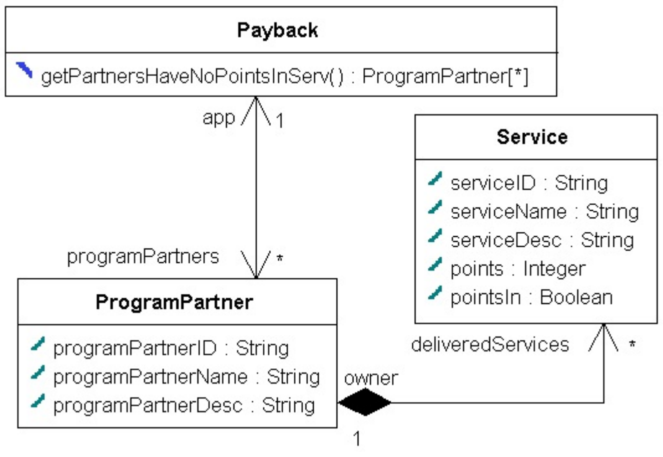


Fig. 2.9.: A simple UML model with complex query operation

As mentioned at the beginning of this section and explained in [OMG10a] in detail, many OCL predefined operations are given, among which are OCL *loop*-operations defined in collection types. These operations are useful and powerful in specifying complex query operations, which are fundamental services in an Enterprise Information System (EIS). The operation *getPartnersHaveNoPointsInServ()* in the class diagram shown in Figure 2.9 has the semantics to gather all the *program partners*, whose all delivered *services* cannot accumulate points. Listing 2.2 shows that this semantics can be specified by an OCL *body*-expression with two loop operations, *select()* and *forAll()* used in a nested way. Listing 2.3 shows the Java implementation of the same operation. It is obvious that the Java counterpart is much more complex than the corresponding OCL expression. This is important for model - compact and precise.

```
body:  self.programPartners->select(deliveredServices
                                        -> forAll(not pointsIn ))
```

Listing 2.2: OCL expression specifying complex query operation

```
ArrayList<ProgramPartner> selectResult_0= new ArrayList<ProgramPartner>();
Iterator<ProgramPartner> itr_0=this.programPartners.iterator();
while(itr_0.hasNext()){
  ProgramPartner itrVar_0= itr_0.next();
  boolean forAllResult_0= true;
  Iterator<Service> itr_1= itrVar_0.getDeliveredServices().iterator();
  while(itr_1.hasNext()){
    Service itrVar_1= itr_1.next();
    forAllResult_0= forAllResult_0 && !itrVar_1.getPointsIn();
  }
  if(forAllResult_0)
    selectResult_0.add(itrVar_0);
}
return selectResult_0;
```

Listing 2.3: Java implementation of the OCL expression in Listing 2.2

As aforementioned, OCL expressions do not change the state of a software system being modeled. That means some important *write* semantics are not involved in OCL. In order to take advantage of using OCL's compact syntax and its powerful predefined operations on the one hand, and on the other hand to involve the missing semantics in an intuitive manner, the OCL is upgraded into a full-fledged AL in this work, which is addressed in Section 3.4 thoroughly.

2.3. Model Driven Software Development

2.3.1. Overview and Typical Applications

As the name suggests, model driven software development, also called *model driven engineering* (MDE) is a technique to develop software, in which models instead of programs are essential artifacts created in the corresponding software development process. As next step, programs implementing the software can be generated from the models automatically [Som12].

In our opinion, the MDE concept can be generalized to describe any software development scenario, in which some well known *aspects* have been specified in an abstract, high-level manner than direct programming, and finally, these specifications (considered as models) are processed by dedicated tools to generate implementation codes. Careful study of such scenarios helps to gain important hints for creating PIMs within MDA-methodology, which will be discussed more detailed in next section.

Scenario 1: The Graphical User Interface (GUI) of a (desktop) application usually consists of a window, which contains different kinds of GUI elements, e.g., menu items, icons, buttons, input fields, etc. According to different platforms, these GUI elements may also be called GUI components or GUI controls. A GUI element usually has parameters, which concern displaying them on the screen correctly. Such parameters are position, size, background- and foreground-color, etc. Most modern OOPLs implement common GUI elements as classes and their important properties as attributes of the corresponding classes. They are deployed in libraries and the programmers can use them directly. However, implementing GUI-layout as source code is still time-consuming. The powerful modern IDEs solve this problem by integrating an additional software component, which supports visual manipulation of GUI elements. The *Form Editor* and its successor, the *WPF Editor* in Visual Studio and *Swing GUI Builder* in NetBeans are typical examples. By using such tools, programmer chooses the required GUI elements from a *toolbox*, which contains all the supported GUI elements in the current context, positions them and edits them visually. The source code implementing GUI-layout is generated automatically.

Scenario 2: Developing data-driven EIS usually involves interactions with relational database. If its *Business Object Layer* is developed in the OO-manner, the *Data Persistence Layer* has to map the class structure of the business object layer into the underlying ER-scheme. This process is called Object Relational Mapping (O/R Mapping). In order to increase the abstraction level and to simplify the O/R mapping, modern software development frameworks are equipped with facilities allowing to specify the necessary information for O/R mapping in the phase of developing the business object layer. For example, the Java Persistence API (JPA) [Gon10] [LB10] uses *annotations* to annotate a normal Java class as entity class, whose instances should be persisted in database, and their *foreign key relations*, whereas the *Propel* [Sch09] for PHP 5 uses an XML-based configuration file to define the entity classes and their relationships. After that the JPA framework processes the annotations to generate correct ER-scheme and a data access interface, whereas Propel generates both the implementation code for the entity classes in PHP 5 and the mapped database scheme.

Scenario 3: Developing parser for a computer language is a time consuming task, so that in most situations a software tool called parser generator will be used to simplify this process. Parser generator allows to move the parser development from time consuming and error prone level of programming to the next higher level of specifying parser in (Extended) Backus-Naur Form (EBNF). Then the parser generator transforms the parser specification into its implementation in one of the common programming languages. For example, the Coco/R [JKU13] can generate LL(k)-parsers based on recursive-decent strategy, whereas GOLD Parsing System [Coo13] generates LALR(1)-parsers based on LALR parsing table.

All the three scenarios above exploit the advantages gained by moving (a part of) software development from low level implementation into high level specification. These advantages can be summarized as follows:

1. The essential logic of a software component is specified in a *domain specific* manner, so that both domain experts and non-programmers can participate in the development.
2. These domain specific developments result in software specifications, which are implementation independent and much more concise and compact than their concrete implementations.
3. A transformation tool or framework, whose internal working is transparent to the end users, is used to transform these domain specific software specifications into implementations based on the implementation requirements.

2.3.2. Model Driven Architecture

The MDE concept can deal with all aspects in a software development process, whereas the Model Driven Architecture concentrates on design and implementation phase [Som12]. As the three MDE scenarios depict, the modeling mechanisms used in MDE can be diverse and more or less tool-specific, whereas in MDA the models should be created in UML and its related concept (e.g., profiles). Figure 2.10 illustrates the essential concepts in MDA from a high-level perspective.

As defined in [OMG03], a *platform independent model* is a view of a system from the platform independent viewpoint. A PIM exhibits a specified degree of platform independence so as to be suitable for use with a number of different platforms of similar type.

A *platform specific model* is a view of a system from the platform specific viewpoint. A PSM combines the specifications in the PIM with the details that specify how that system uses a particular type of platform.

Model *transformation* is the process of converting one model to another model of the same system. Note that the original demonstration of the MDA pattern was given in [OMG03] in a more generic style, in which the parts known as *transformation specification* in Figure 2.10

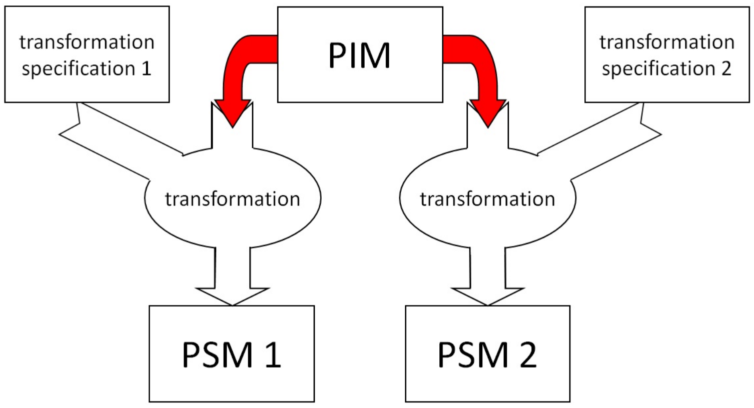

transformation specification 1
PIM
transformation specification 2
transformation
transformation
PSM 1
PSM 2

Fig. 2.10.: High-level perspective of MDA concept

is still a blank. Representing MDA concept in this manner is intended to be suggestive. The PIM and *other information* are combined by transformation to produce a PSM. There are many ways in which such a transformation may be done, leaving the tool designer to develop their own (innovative)-way realizing the transformation.

2.4. Related Work

2.4.1. Methodology and Tool Support for Efficient Modeling

Since MDA concept was launched in 2001 [OMG03], a lot of approaches and their tool supports have been developed. As introduced in Section 1.2, the both key issues, which MDA tries to solve, are *efficient modeling* and *model transformation*. In this section, a brief summary of researches in the both fields and their tool supports is given, in order to provide an overview of the technological states nowadays.

As well known, if a model contributes to a problem solution, that model must be less complicated than a solution itself. Further more, the model has to be correct for the problem solution and easy to understand. In the software development, this kind of model can be created by concentrating only on the problem domain, leaving other IT-infrastructure details unconsidered at first. Typical examples are *Entity-Relationship* (ER) diagram used to model data entities involved in a relational data base and (E)BNF-notations as well as *syntax directed translation* [Aho+08] used to specify compiler logic. Both examples originate in the same philosophy: *the significant characteristics of problem domain being considered must be well studied and then modeled in a concise manner independent of a concrete implementation.* Following this idea, the efficient modeling means domain specific modeling, along with MDA, means, the PIM or at least, a large part of PIM must be domain specific. To achieve the domain specificity, either *domain specific language* or UML with *domain specific profiles together with reusable design platform models* can be used.

As introduced in Section 2.2, DSL is a computer language with special purpose, which may have no relationship with UML, e.g., SQL for database related operations, DSL for parallel computing [Suj+11], etc. Other DSLs can be considered as MOF-compliant, e.g., the *Scene Structure and Integration Modeling Language* (SSMIL) [Vit08] for modeling 3D-applications, and the OCL. As domain specific profiles, there are e.g., profiles for GUI modeling [Bla04], profiles for modeling Service Oriented Architecture (SOA) based applications [HLT03] [Lóp+07], and profiles for modeling context aware web applications [Kap+09] as well as the OMG EDOC profile [OMG04] for developing enterprise applications.

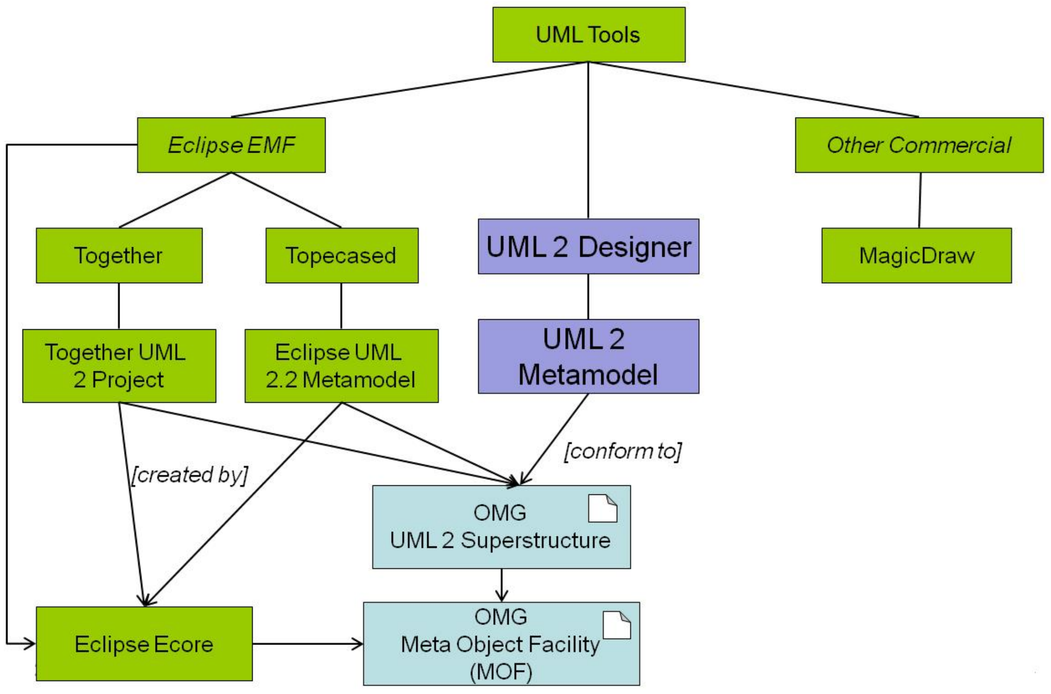


Fig. 2.11.: The tool support for efficient modeling

No matter which method is used, necessary tool support is indispensable. The presented work concentrates on efficient modeling in UML and the transformation of UML models. Thus, only the diverse tools in terms of UML modeling are concerned. As Figure 2.11 shows, the tools are classified into three groups:

1. Eclipse based modeling tools
2. Our own UML tool - the UML 2 Designer
3. Diverse commercial tools

The most academic research projects concerning MDA are based on the first group of modeling tools. Because Eclipse [Ecl13] is a great open source project, whose purpose is to provide a highly integrated tool platform. Thus, eclipse is far more than a well-known IDE for Java programming language, but a framework to implement development tools for diverse programming languages and even modeling languages [Ste+09]. The *Eclipse Modeling Framework* (EMF) is its modeling framework that exploits the facilities provided by Eclipse [Ste+09]. The EMF is not conceived just as a UML editor. Within EMF arbitrary DSLs can be defined together with their editors generated. With the OMG four layered meta-model architecture introduced in Section 2.1.2 in mind, this flexibility can only be obtained by providing an MOF-like meta meta level modeling facility. That is the Ecore [IBM12] API. For this work, the only important thing is to understand that Ecore API can be considered as an EMF specific implementation of OMG MOF standard as illustrated in Figure 2.11. Given Ecore API, the UML meta model can be created. Figure 2.11 shows two EMF-based modeling tools, each of which has its own meta model project conforming to Ecore. Along with this strategy, even the abstract syntax (meta model) of Ecore-compliant DSL can be created by using Ecore API. The SSMIL-DSL is a typical example.

In the middle branch of Figure 2.11 are our proprietary CASE-tool *UML 2 Designer* and its metamodel project. Both of them belong to our research project *UML-TEst Front-End* (UTEFE)

[SBL04], which was financed by the Bayer Business Services GmbH and consisted of 6 project phases. The primary motivation of UTEFE project was that in early 2004, when the UTEFE project was started, the most CASE-Tools relied on meta model of either UML 1.x version or a mixture of 1.x and 2.0 version of UML, and the other research project *GEneration of TEst CAses* (GETECA) [Ste+02] would exploit the powerful features of UML 2.0 standard, especially in behavioral modeling. Thus, a completely UML 2 compliant CASE-tool was developed under leading of Prof. Steinbach in our institute. With years of continuing development, the UML 2 Designer has become a full-fledged CASE-tool for both software and business process modeling and has served as the front end in all of our UML-centric research projects.

The rightmost branch of Figure 2.11 represents commercial CASE-tools, among which the *MagicDraw* is given as a concrete example. The benefit of using such powerful commercial CASE-tools is obvious, whereas the disadvantage is the relatively expensive licensing fee and the lack of full control over the tool in terms of modification and extension.

2.4.2. Methodology and Tool Support for Model Transformation

As introduced in Section 2.3.2, the primary objective of model transformation in the context of MDA is clear, whereas the possible realization is given in a suggestive manner, to encourage development of innovative methodologies. Generally speaking, there are two kinds of model transformations, namely the *Model to Model* (M2M) transformation and the *Model to Text* (M2T) transformation as shown in Figure 2.12. Given a PIM modeled with the technology and tool addressed in the previous paragraph, both M2M and M2T transformations are usually involved in one MDA process, transforming an abstract, domain specific PIM into one of the required PSMs and then into the source code of the target platform.

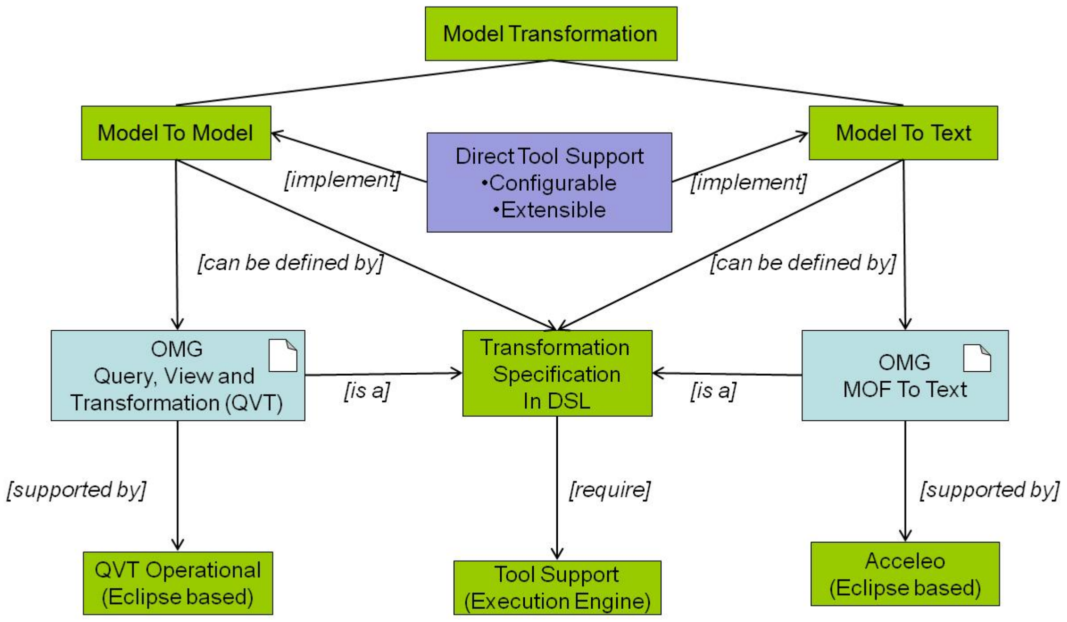


Fig. 2.12.: The tool support for model transformation

For both the M2M and M2T transformations the regular solution is to develop some kind of appropriate DSL, i.e., the *model transformation language*, with which the transformation logic can be expressed relatively high-level. For both cases there are even recommended standards from OMG, namely, the QVT [OMG11b] for M2M transformation and MOF To Text [OMG08] for M2T transformation. Since the M2M transformation is considered as essential and complicated in the the entire MDA process, the most MDA researches fall into this branch to work out efficient M2M transformations. For example, besides QVT, the *Atlas Transformation Language* (ATL) [JK06b] is also widely accepted, which shares almost the same idea with QVT and can be aligned to QVT from the perspective of language architecture [JK06a]. Because such leading M2M transformation DSLs are usually text-based (the QVT Relations [OMG11b] is equipped with a graphical concrete syntax, but the primary usage is textual). Researchers like [GMO09] consider this phenomenon as a paradox that *the model-driven community promotes the usage of models instead of textual code, while the same community dominantly uses textual code to define the model transformations.* As an answer of the question above, M2M transformations relying on graphical notations [KBC03], based on *Triple Graph Grammars* [LG08] [GLO09] have also been developed.

As illustrated in Figure 2.12, all the aforementioned methodologies can be considered as high-level specification of model transformation logic into some kind of DSL, either textual or graphical. Then, such a specification is compiled into the model compiler for M2M transformation or code generator for M2T transformation implemented in normal programming language. As tool supports, there are *QVT Operational* for QVT Operational Mapping and *Acceleo* [Acc13] for MOF To Text respectively. Both of them belong to the large Eclipse ecosystem. Specifying model transformation in this way requires knowledge of the meta models of both the source and target models. Although the resulted specification can be reused, even a minor modification involved in the transformation concerns changing in DSL-source code or DSL-source model in the case of graphical DSL and the following recompiling.

As shown in the middle of Figure 2.12, the both M2M- and M2T-transformations can be implemented in a tool, which, according to our research, is a model transformation framework. Because the fundamental of model transformation is well known, the model transformation can be decomposed into a model mapper hierarchy, with the root model mapper implementing the most fundamental transformation common among all target platforms and each leaf model mapper dealing with transformation issues specific to a target platform. Within this transformation framework, the input model is completely platform independent and created based on a well designed, highly abstract design platform model and a powerful action language rooted in widely used OCL. To map model elements from design platform model onto a target platform, an XML-based *mapping configuration file* is used, which leads to *configurability*, whereas novel model mappers can be added into the framework with their own mapping rules, which support *extensibility.* The former will be addressed in Section 4.3.4 and the latter in Section 4.3.3 in the presented work.

3. The MOCCA Modeling Framework

3.1. Scope and Design Philosophy

This chapter addresses the first issue raised in Section 1.2 - compact and complete modeling for MDA. The first attempt towards solving this problem is to fix a reasonable scope for modeling. As explained in [Som12], a software development process usually encompasses many different *process activities*, from initial conception to final deployment and maintenance. Regardless of which process model is used, there are four fundamental activities common to all the process models:

1. *Software specification* defines the functionality of the software and the constraints on its operation.
2. *Software design and implementation* produces the software, which meets the specification.
3. *Software validation* ensures that the produced software does what the customer wants.
4. *Software evolution* deals with changing customer requirements.

We agree with the author of [Som12] that MDA concentrates on software design and implementation. Because the ultimate goal of MDA is to generate executable applications from semantically correct and complete models. As key design philosophy, the data-centric, procedural paradigm usually based on *transaction script* [Fow+02] design pattern is excluded. Taking into account these considerations the models created within the MOCCA modeling framework are defined as *object oriented design model*, which hides any implementation specific details related to one concrete platform on the one hand, but involves both structure and behavior completely on the other hand. Such a model is called *Design Model* (DM) according to MOCCA terminology. In contrast to an OOA-model, a MOCCA DM is built under some technical constraints given in the MOCCA DPM, which will be studied in detail in next section.

Other common activities mentioned above can also be supported by model driven technology, i.e., model driven requirements engineering, model-based test case generation etc., which do not belong to the scope of this thesis.

The next step concerning scope and design philosophy is to choose appropriate UML diagram types to create MOCCA DM. There are thirteen different diagram types coming with UML 2 standard, which cover almost all the aspects of software modeling. The baseline to select diagram can trace back to the first and second sub-issue of Section 1.2, which can be divided further into the following four aspects.

Complete structure modeling

A MOCCA design model contains the overall architecture of a software application. Such an architecture is usually decomposed into *layers* or *components*, which can be physically organized as packages. Package consists of classes, interfaces, data types and their relationships. In order to make discussion simpler, an umbrella concept *classifier* is used to refer to the common model elements in terms of class diagram as a whole, namely class, interface and data type. Based on this narration, the UML *package diagram* and *class diagram* are inherently good choices for creating design models. Because UML class diagram also supports the package concept. Hence, only *class diagram* is used for modeling structure.

The UML *component diagram* seems to be a good candidate to model software structure. Because component diagram shows the cooperating modular parts - the components - in terms of provided and required interfaces. Moreover, the white-box representation [RQZ07] of a component can involve deployment artifact of the actual component. But according to our experience, the structure modeled in component diagram is yet too coarse-grained to transform into implementation automatically. It is clear that with the logic of MOCCA model mapper and mapping rules a significant amount of additional classifiers and their involved elements can be generated from design model into target model. However, there are always fine-grained details like properties and operation signatures, which belong to core application structure, have to be modeled exactly in class diagram. The *component diagram* is not used to create design model in this thesis. However, certain ready-to-use software *component* like parser for arithmetic expressions can of course be used to create high-level design model in the class diagram by calling the provided interface(s) of such a component.

Compact structure modeling

Compared to organize software structure directly on the code base, modeling structure in class diagram is indeed more compact, in which many classifiers scattered in different source files on the code base can be involved in a single class diagram on the one hand, the different relationships among those classifiers can also be represented clearly on the other hand. But there are yet problems coming with class diagram. Essentially speaking, class diagram merely shifts the textual representation of classifiers into graphical representation, without compressing the information itself. That means, the classifiers as well as their contents, which have to be edited on the code base, must be modeled in class diagram, too. With increasing number of classifiers and relationships among them, the compactness in class diagram will be lost more or less.

To achieve the compact structure modeling the information involved in class diagram should be compressed in such a way that the number of classifiers and their relationships should be minimized without losing structural completeness for model transformation. Inspired by the analysis in Section 1.1, the classifiers in terms of cross-cutting concerns and plumbing can be hidden from the core concerns. In this work, two strategies are adopted for this purpose:

1. Similar to the class library of modern OOPLs, classifiers related to cross-cutting concerns and plumbing can be abstracted and adopted into a reusable and well documented modeling library, which is actually the MOCCA DPM. Usually, if a classifier in terms of core concerns has an association with DPM data type, instead of a UML association, the DPM data type is used to define a normal property in a class.

2. There are situations, in which cross-cutting concerns and plumbing should be involved into DM by using high-level expressive constructs, to hide unnecessary details. Sharing the common idea with the first example in Scenario 2 of Section 2.3.1, the UML stereotypes and their tagged values can decorate any UML element including classifier with additional (complex) information in a clean way. Stereotypes can be interpreted well by model mapper. In this work they are defined in dedicated profiles and launched in DPM as well.

Complete behavior modeling

Because the ultimate objective is to generate full-fledged applications implemented in one of the OOPLs, complete behavior modeling means that the method implementations must be generated from their behavioral models. As classified in Figure 2.1, there are seven diagram types concerning behavior modeling, four of which belong to the subclass *interaction diagram*. As the name indicates, the interaction diagrams are good at illustrating message exchanges between interaction participants from different points of view. For methods with many interactions, i.e., events trigger, it is clear that the *sequence diagram* is able to model the entire behavior. However, as proven in [Did10], the sequence diagram is not the wise choice for methods concerning intensive computing. The *state machine diagram* has been used to model behaviors for embedded systems. However, similar to sequence diagram, it is not and should not be a one-for-all solution. The sole diagram type that can be used to model method implementation completely is *activity diagram*, which supports all the three control flows as well as elementary semantics like local variables. As an early attempt, in my master thesis [Lia08], the activity diagram was used together with OCL to model computation intensive methods, e.g., sorting algorithms, for code generation. Based on our experience, for some complex computation, the activity represented in diagram seems more complex than its implementation code. Hence, the activity diagram fulfills the requirement of complete behavior modeling but not the compactness of such a model.

Compact behavior modeling

Continued with the problem above, none of the UML behavior diagrams satisfies the requirements of both complete and compact behavior modeling. Inspired by the *pseudo code* [Dal13], which is primarily used to formulate algorithms concisely, the decision is made to use high-level textual action language (AL) for behavior modeling in design model. As illustrated in Figure 2.6, the overall feasibility of using AL is guaranteed by UML metamodel. The metaclass *OpaqueBehavior* encapsulates AL statements or expressions and associates to an operation as the model of its implementation. In this work the OCL is chosen as AL for its compactness, which can be easily aware by comparing Listing 2.2 and Listing 2.3. The small number of missing semantics in terms of changing system states have been involved into original OCL by additional language constructs, so that the upgraded OCL, namely XOCL has become a full-fledged AL for both complete and compact behavior modeling.

3.2. Overview of Platforms and Models

Up to now several important platforms and their models have been mentioned and even defined in the previous chapters. However, the lack of summarizing these models as well as their relationships prevents us from understanding the overall design philosophy and the application of these models. To give this overview, it is necessary to re-mention the methodology used in this work - the platform based MDA approach. The MDA has been introduced in Section 2.3.2. The *platform based* characteristic roots in the work of [SFB05] and similar to the research of [WJ04], that means the development of applications is based on platforms, whereas different platforms are used for design, implementation and deployment. Taking into account these considerations, Figure 3.1 illustrates all the models within the MOCCA modeling framework.

The horizontal separation of Figure 3.1 conforms to the MDA paradigm, in which both PIM and PSM concepts are defined. Due to the platform-based strategy, Figure 3.1 involves further more a vertical separation, whose upper part can be summarized as application model, whereas lower part as platform model. A *platform model* provides a set of technical concepts, representing the different kinds of parts that make up a platform and services provided by that platform. An *application model* describes a software system using the services provided by the underlying platform. Both application models and platform models can be classified into platform independent

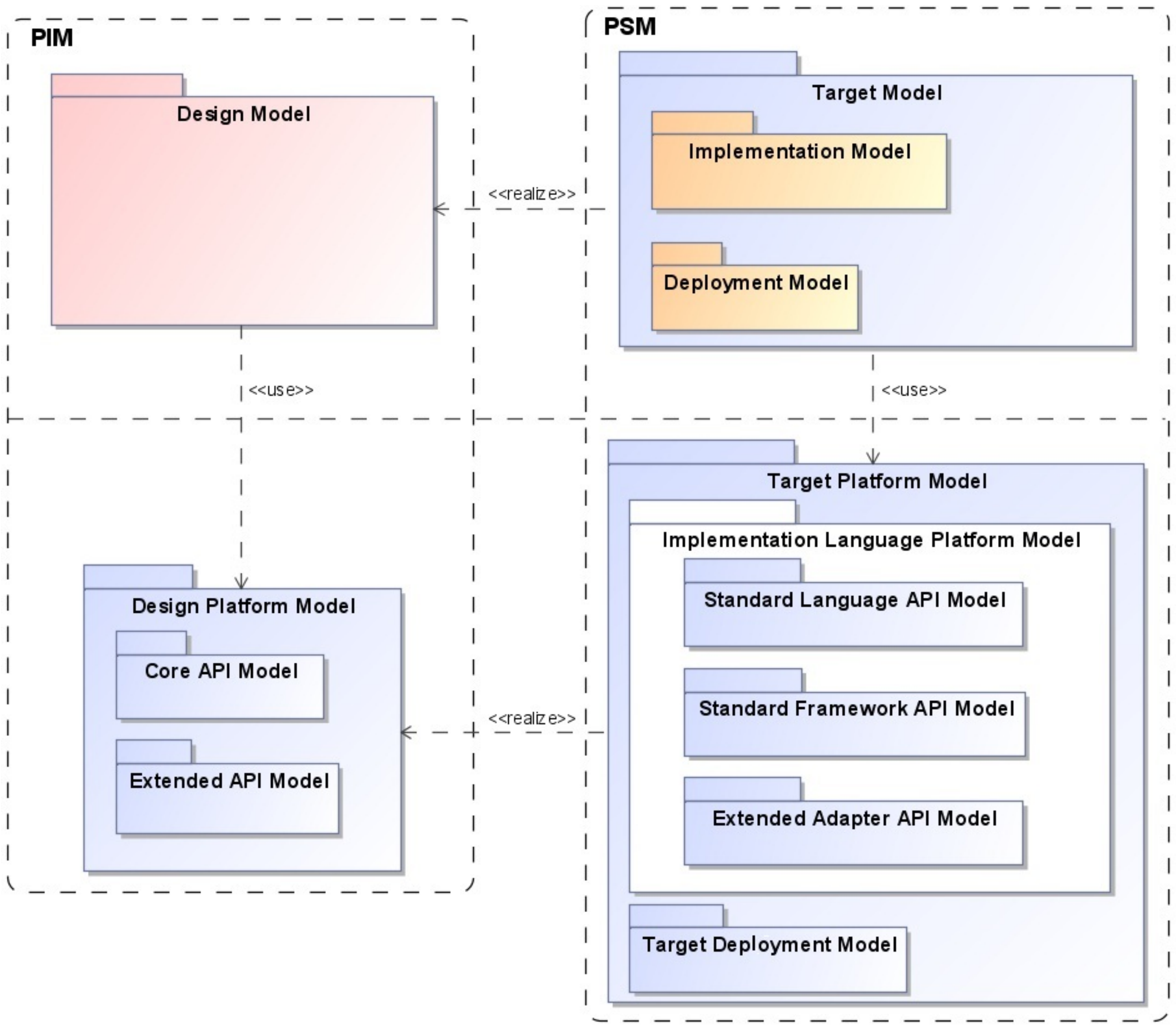


Fig. 3.1.: Overview of different models within MOCCA modeling framework

and platform specific category.

Design Platform Model

According to the MOCCA terminology, the platform model residing in the PIM branch is called design platform model. A *design platform* is an architecture platform that comprises a set of ready-to-use data types, their relationships, and constraints as well as important meta-data. The MOCCA DPM is a UML model that represents such a design platform. In the current MOCCA DPM all the ready-to-use data types and their relationships are modeled using normal UML data types and relationships in separate class diagrams, constraints are given by OCL expressions and meta-data are represented by UML profile and stereotypes.

As shown in Figure 3.1, the DPM itself consists of two parts. The *Core API model* refers to basic data types common to most applications. Such data types are typically primitive data types, collection data types, data types concerning IO-operations, etc. Further more, many stereotypes backed by mature design patterns are also involved into this category. Sharing the similar idea of the *Mercator-Framework* [WJ04], the *Extended API Model* serves as extension point to involve domain specific functionality for a special group of MOCCA users, e.g., banking branch or 3D-development. The crucial requirements for DPM model elements can be summarized as follows:

1. for each DPM data type as well as its attributes and operations there must be either a mapping rule defined in *Mapping Configuration File*, or a piece of core algorithm implemented as firmware in a corresponding platform specific model mapper,
2. for each meta data defined in the form of stereotype there must be an appropriate core algorithm in model mapper to interpret such stereotype in a platform specific manner.

Target Platform Model

A Target Platform Model describes a concrete technical platform, on which the implementation language and its API are provided on the one hand, on the other hand the mechanism to deploy a generated application should also be given. Based on this separation, the MOCCA TPM is divided into Implementation Language Platform Model (ILPM) and Target Deployment Model (TDM). As the name suggests, the MOCCA ILPM abstracts different APIs in the same target language, which can be divided further into three categories:

1. Standard Language API Model
2. Standard Framework API Model
3. Extended Adapter API Model

The *Standard Language API Model* encapsulates the latest language status as documented by the language vendor. Based on the peculiarity of the target language, the UML model representing this category of ILPM can be very similar to the language API itself or some sophisticated restructuring has to be done to smooth some kind of non-OO characteristics in target language. For Java programming language, both the JSE-API [Ora12] and JEE-API [Ora11a] can be seamlessly modeled in UML class diagram, because the Java language and its API share almost the same OO-principle with UML. Even the special language construct *annotations* and their *values*, which are used intensively in JEE applications can be modeled as stereotypes and tagged values.

The *Standard Framework API Model* sounds a little confused. People also call the Java collection API as collection framework, Java Persistence API (JPA) [Ora11a] as Java persistence framework, etc. If such frameworks belong to standard language API, they should also be considered as parts of the language itself. Hence, according to our classification, all the APIs, frameworks developed and maintained by the original language vendor are considered as standard language API. *Standard Framework API* means widely adopted (commercial) framework written in the same target language but developed and maintained by third-party institution other than original language vendor. For Java language, the standard framework API model can be used to encapsulate the famous Spring Framework API [Spr13].

The *Extended Adapter API Model* is used primarily to abstract some non-OO aspects in target language like ABAP. As the name suggests, the idea coming with this kind of ILPM sub-model is based on the *adapter design pattern* [Gam+95] [FF04].

The *Target Deployment Model* is reserved for future use and not studied thoroughly in this thesis. Because modeling and generating deployment plan for modern application, especially, for distributed enterprise applications, is as sophisticated and complicated as code generation. For example, how to deploy each component of a generated JEE-application into appropriate JEE container by using whether the *ANT-script* [ANT13] or even more powerful *Maven Project Object Model* [Mav13] file can be a challenging research topic by itself.

Design Model

The *Design Model* is a platform independent UML model representing the main application logic. It contains model elements defined by application modelers or provided in DPM. The XOCL expressions specifying behaviors are also stored in design model. Design model is the most important input for the MOCCA model compiler.

Target Model

The *Target Model*, which can be further divided into *Implementation Model* and *Deployment Model*, can be considered as the product or output of a concrete model mapper. According to the current status of MOCCA, the implementation model saves only structural information given by UML meta-model, but not the graphical information for visualizing a UML model. The UML model elements involved in an implementation model represent one-to-one mirror of an OOPL-implementation. Similar to target deployment model, the deployment model is reserved for future work.

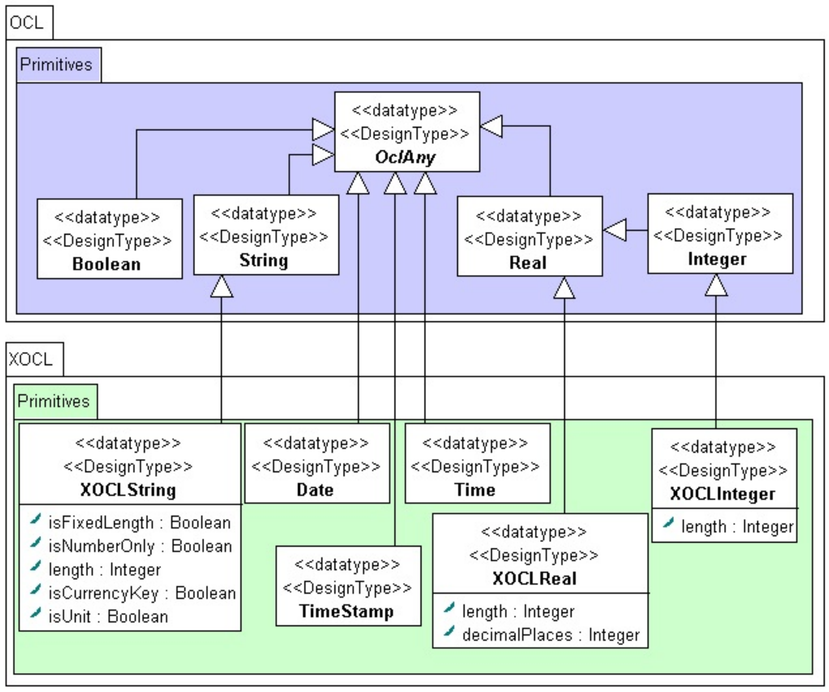

OCL
Primitives
<<datatype>> <<DesignType>> OclAny
<<datatype>> <<DesignType>> Boolean
<<datatype>> <<DesignType>> String
<<datatype>> <<DesignType>> Real
<<datatype>> <<DesignType>> Integer
XOCL
Primitives
<<datatype>> <<DesignType>> XOCLString
isFixedLength : Boolean
isNumberOnly : Boolean
length : Integer
isCurrencyKey : Boolean
isUnit : Boolean
<<datatype>> <<DesignType>> Date
<<datatype>> <<DesignType>> Time
<<datatype>> <<DesignType>> XOCLInteger
length : Integer
<<datatype>> <<DesignType>> TimeStamp
<<datatype>> <<DesignType>> XOCLReal
length : Integer
decimalPlaces : Integer

Fig. 3.2.: Primitive data types involved in the MOCCA DPM

3.3. The MOCCA Design Platform Model

3.3.1. The Primitive and Collection Types

The design platform model is the foundation stone for building design models. Similar to developing applications in programming languages, primitive data types like *integer* and *string* as well as collection type like *vector* are ubiquitous in modeling, too. Instead of inventing such types from the ground, the primitive data types and collection types defined in OCL are involved in MOCCA DPM on the one hand. Other useful elementary data types are selectively adopted into our XOCL extension to deal with diverse modeling requirements on the other hand. In fact, relying on open standard is one of the fundamental design philosophies of MOCCA DPM. Adopting this strategy helps potential users grasp our technology, saves time to document these types and their operations.

Figure 3.2 summarizes the primitive data types of MOCCA DPM in a class diagram. To make clear difference between primitive data types defined in OCL standard and primitive types involved in XOCL extension, they are grouped into corresponding packages with intuitive names. As shown, the *OclAny* is modeled as the supertype for all the other types represented here. In fact, as explained in [WK03], it is considered to be the supertype of all classifiers in both DPM and DM. That means, all predefined types and all user-defined types inherit the features of *OclAny*. Thus, the *OclAny* finds its common usage like *java.lang.Object* class in Java and *System.Object* class in .Net. Detailed information about *OclAny* can be found in [WK03], because its semantics and predefined operations are preserved in DPM completely.

```
context XOCLString
inv: self.isFixedLength = false
inv: self.isNumberOnly = false
inv: self.length.oclIsUndefined()
inv: self.isCurrencyKey = false
inv: self.isUnit = false
```

Listing 3.1: OCL invariants specifying default domain of XOCLString

The primitive types *Boolean*, *Integer*, *Real* and *String* preserve their original semantics and predefined operations. As described in [WK03], the *Boolean* type can only hold one of the two values: *true* or *false* and supports common logical operations like *or*, *and* and so on. The *Integer* and *Real* represent the common mathematical concepts respectively. Because OCL is a modeling language, there are no restrictions on their values. However, the exact domain of the both types can be retrieved by consulting the corresponding mapping rules for them onto a concrete target platform. For example, if a design model class, which contains a property defined as DPM *Integer*, will be mapped onto Java Platform Standard Edition (JSE), and further more, there is mapping rule given in JSE mapping configuration file, which stipulates that the DPM *Integer* is mapped as Java primitive data type *int*. In this way, the non-restricted DPM *Integer* and *Real* will be constrained by concrete data types of target platform, onto which they will be mapped. The *String* type represents a sequence of printable characters and provides several predefined operations such as *concat()*.

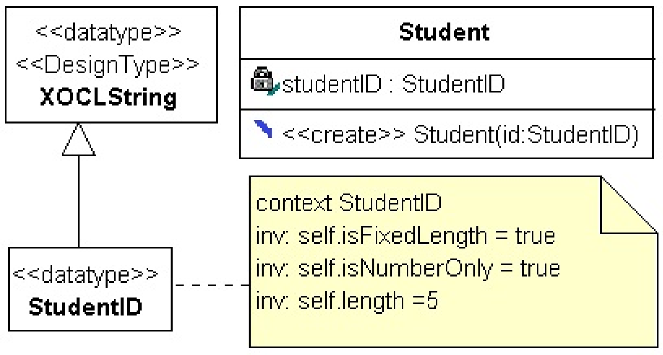


Fig. 3.3.: A user-defined primitive type with strict domain

In practice, there are occasions, where domain of primitive data types should be defined exactly. For this purpose, three additional primitive data types derived from original OCL primitive types, namely, *XOCLString*, *XOCLReal* and *XOCLInteger* are given in Figure 3.2. As their prefix indicates, they belong to the XOCL extension. These types take over the original semantics by inheritance and add extra properties for restricting their domains.

By default all the three XOCL extended primitive types preserve the original semantics in OCL. For example, the domain of *XOCLString* is defined via OCL-invariant as shown in Listing 3.1. If a string-based type with special domain must be used in the model, it can be easily constructed by inheriting *XOCLString* and defining invariants on it. Figure 3.3 illustrates an example. A string-based data type called *StudentID* is defined as five-digit, number-only, which can be used to define typed elements such as the *studentID* property and the corresponding constructor parameter in the *Student* class. Primitive data types defined in such a way usually have concrete semantics and strict domain definitions, which are important for some kind of applications.

The XOCL primitive type *Date* is used to model time information defined by given year, month and day. The *Time* is used to model time information defined by given hour, minute and second. The *TimeStamp* represents the current system time.

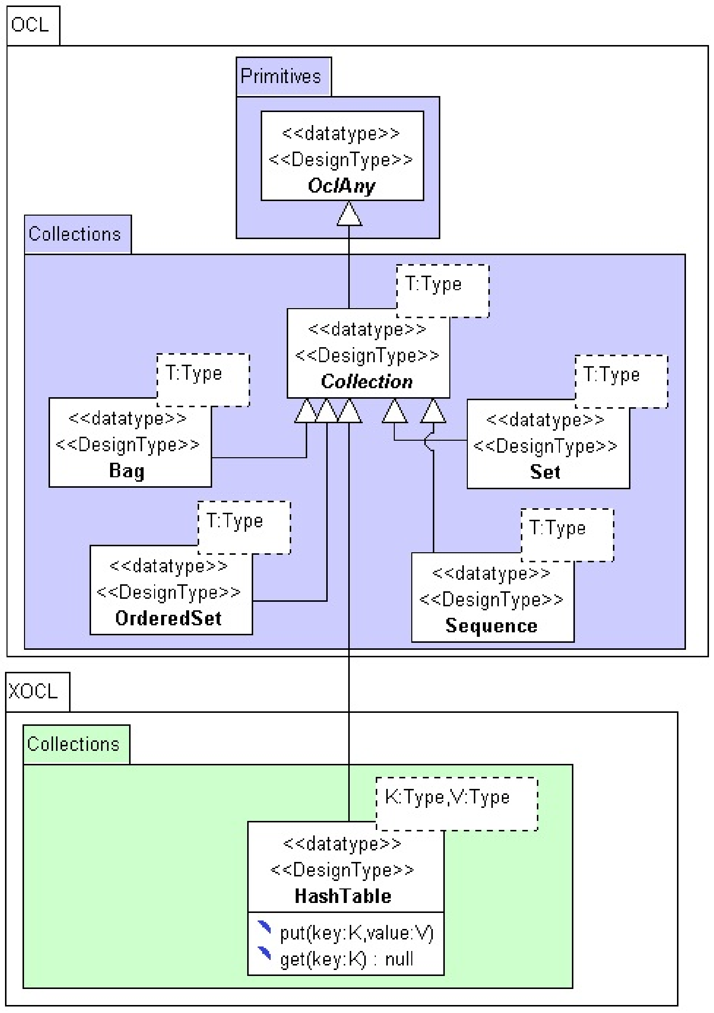

OCL
Primitives
<<datatype>> <<DesignType>> OclAny
Collections
T:Type
<<datatype>> <<DesignType>> Collection
<<datatype>> <<DesignType>> Bag
<<datatype>> <<DesignType>> Set
<<datatype>> <<DesignType>> OrderedSet
<<datatype>> <<DesignType>> Sequence
XOCL
Collections
K:Type,V:Type
<<datatype>> <<DesignType>> HashTable
put(key:K,value:V)
get(key:K) : null

Fig. 3.4.: Collection types involved in the MOCCA DPM

Figure 3.4 shows the hierarchy of collection types defined in MOCCA DPM. Similar to primitive types, the original OCL collection types preserve their semantics and operations. They have been documented in [WK03]. One more collection type, namely the *HashTable* is added to extend standard OCL collection types. A *HashTable* represents a dictionary-based collection, which maps keys to values. All data types, except *OclVoid* can be used as key or as a value. Besides taking over the operations from the OCL abstract collection type *Collection* by inheritance, two more operations *put(K, V)* and *V get(K)*, which are essential for dictionary-based collections, are defined.

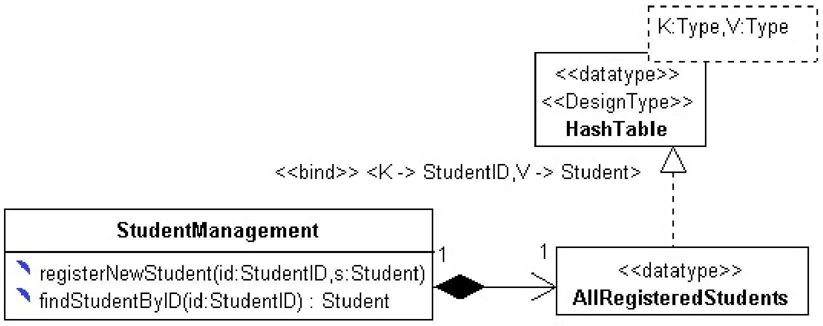

K:Type,V:Type
<<datatype>>
<<DesignType>>
HashTable
<<bind>> <K -> StudentID,V -> Student>
StudentManagement
registerNewStudent(id:StudentID,s:Student)
findStudentByID(id:StudentID) : Student
1
1
<<datatype>>
AllRegisteredStudents

Fig. 3.5.: Association-End interpreted as HashTable

As introduced in 2.2.2, OCL collection types are generic, or in other words, template-based types. They must be concretized either by defining local variable in expressions or by interpreting additional information wrapped around a typed element. Especially, for navigating via association-end, a concrete *HashTable* can be modeled as shown in Figure 3.5, where a class *StudentManagement* composition-associates with a data type *AllRegisteredStudents*, which concretizes the generic *HashTable* via *TemplateBinding* relationship. The *TemplateBinding* substitutes the template parameters *K* and *V* with concrete types *StudentID* and *Student* respectively. It is worth noting that the role name, or in other words, the name of the association-end is not given explicitly. In fact, at the time of model transformation, the name of concrete hash table, *AllRegisteredStudents* in this example can be reserved as role name. Because in most popular OOPLs an anonymous concrete hash table is used to define property in a class. In such a situation, an identifier must be chosen to name this property. To illustrate this mechanism, a Java class, which implements the model of Figure 3.5, is shown in Listing 3.2. It is easy to find that the property *AllRegisteredStudents* is defined by an anonymous concrete hash table with both template parameters substituted.

```
public class StudentManagement
{
  private Hashtable<StudentID , Student> AllRegisteredStudents;

  public void registerNewStudent(StudentID id , Student s);
  public Student findStudentByID(StudentID id);
}
```

Listing 3.2: A Java class implementing the model of Figure 3.5

As shown in Figure 3.4, the XOCL *HashTable is an* OCL *Collection* with additional operations in terms of hash table semantics. In concrete syntax, *HashTable* is used as OCL collection type, that means defining local variable in expressions like

- *lvar:HashTable(StudentID, Student)*

and calling operation on it like

- *lvar->put(sid, studInst).*

3.3.2. The Platform Independent GUI Toolkit

GUI-based applications are ubiquitous, which cover classical desktop applications, rich Internet applications as well as applications running on smart devices. In such an application a user interacts usually with a window containing different kinds of GUI elements, which are typically (context-)menu items, icons, buttons, input fields etc. According to different platforms, these GUI elements may also be called GUI components [HC08] or GUI controls [Tro07]. Since the GUI is a significant part of most applications, it should also be modeled using UML. However, it is by no means always clear how to model user interfaces using UML [SP01]. It is not easy to identify how GUI elements are supported in UML models. [SP01] and [Bla04] summarized several common GUI modeling problems when using UML. To solve these problems, either additional diagram notations have been added to the original UML [SP03] or novel UML profile was developed to enhance GUI modeling using UML [Bla04]. However, all the aforementioned researches constrained in extending UML with the ability to create UI model completely, but do not consider UI modeling in the context of MDA. Based on the contributions of [SP01] [SP03] [Bla04] we try to find the modeling solution of GUI in UML to create complete and compact PIM on the one hand, on the other hand to transform the platform neutral GUI toolkit involved in the DPM onto different target platforms efficiently in this work.

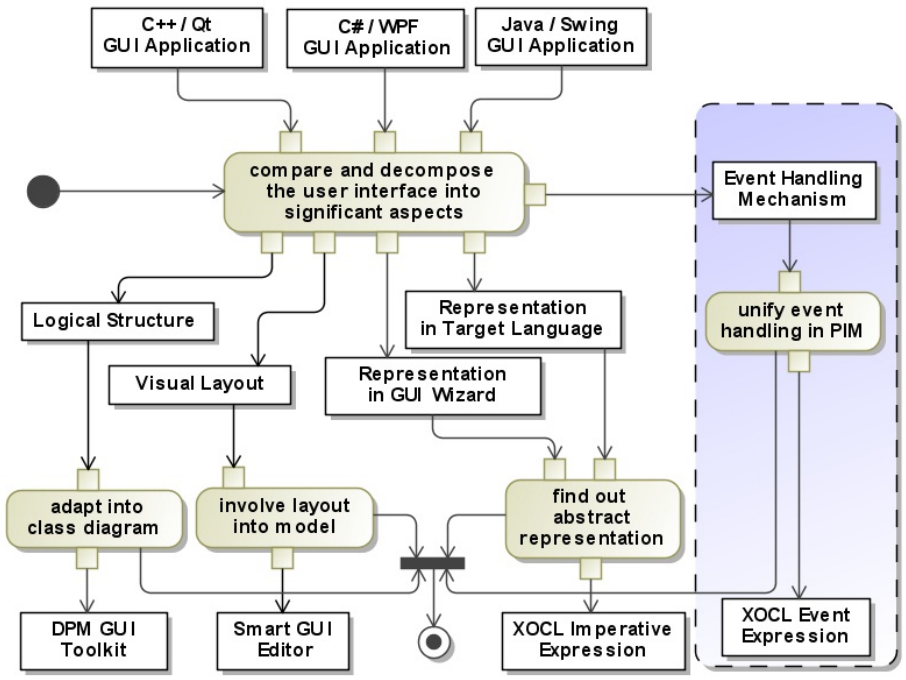

C++ / Qt GUI Application
C# / WPF GUI Application
Java / Swing GUI Application
compare and decompose the user interface into significant aspects
Event Handling Mechanism
unify event handling in PIM
Logical Structure
Visual Layout
Representation in GUI Wizard
Representation in Target Language
adapt into class diagram
involve layout into model
find out abstract representation
DPM GUI Toolkit
Smart GUI Editor
XOCL Imperative Expression
XOCL Event Expression

Fig. 3.6.: The work-flow, in which common UI aspects were identified and the platform independent modeling solutions were found

The most important activities that have been performed in this work to find out the solution of platform independent GUI modeling can be illustrated in an activity diagram shown in Figure 3.6. The region with light-blue background color will be addressed in the next section.

A simple GUI application implemented in C++ with Qt [Qt13] as GUI library, in C# with WPF [Tro07] as GUI library as well as in Java with Swing [HC08] as GUI library is shown in Figure 3.7. As starting point, all the three concrete implementations were compared and decomposed into five important aspects, which are common in developing GUI applications in different target OOPLs.

The *logical structure* of GUI means the entire collection of GUI elements and their *containing-*

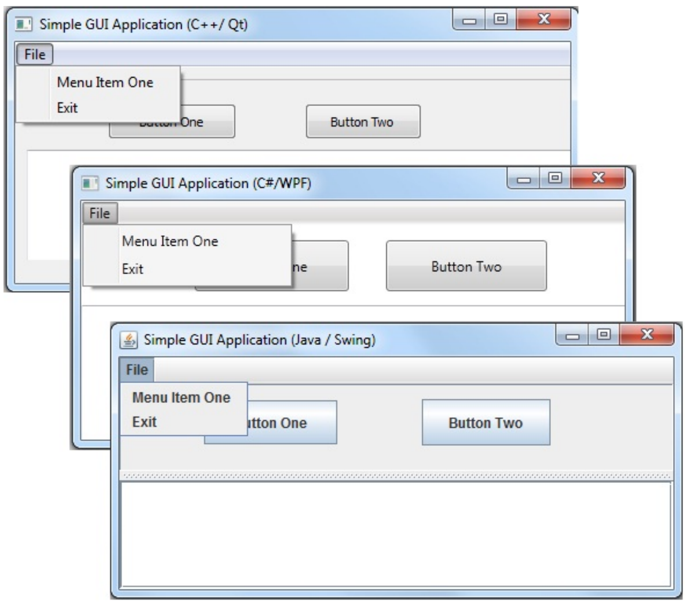


Fig. 3.7.: A simple GUI-based application created in ISO C++ with Qt as GUI library, in C# with WPF as GUI library as well as in Java with Swing as GUI library respectively

relationships. For example, the out-most window contains a menu-bar, which contains a menu labeled as *File* containing further two menu items and so on. Most modern OOPLs implement common GUI elements as classes and their important properties as attributes of the corresponding classes. They are deployed in libraries and can be used in certain languages. Hence, the programmers can use them directly. In the current example, the top level window is implemented in Qt library as *QMainWindow*, in WPF library as *Window* and in Swing library as *JFrame* respectively. For other common GUI elements, the same rule remains.

Independent of concrete GUI libraries, the logical structure of the GUI application above can be modeled using pseudo GUI elements in UML class diagram as shown on the left hand side of Figure 3.8. It is obvious that the GUI structure shown here is a tree rooting in the pseudo GUI element *MainWindow*. The *composition associations* between *MainWindow* and *Menu*, *MenuItem*, *Container*, *TextOutput* as well as *Button* model the creation semantics that such GUI elements are created and only exist within the life cycle of a *MainWindow* object. Other associations build the tree structure of this GUI.

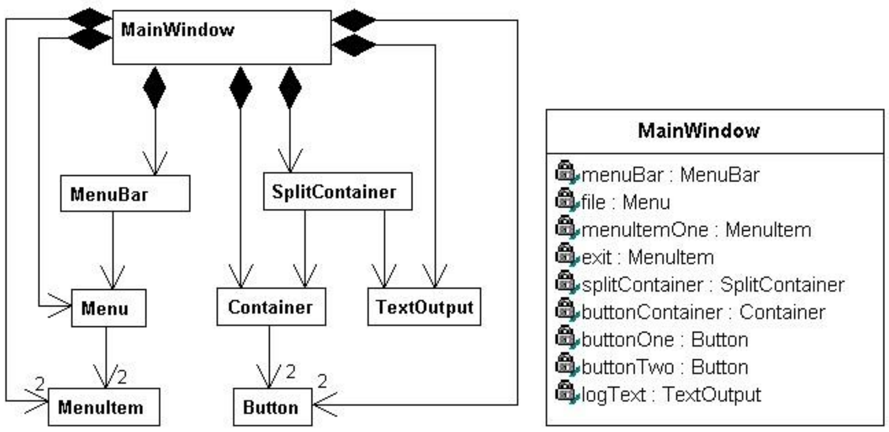
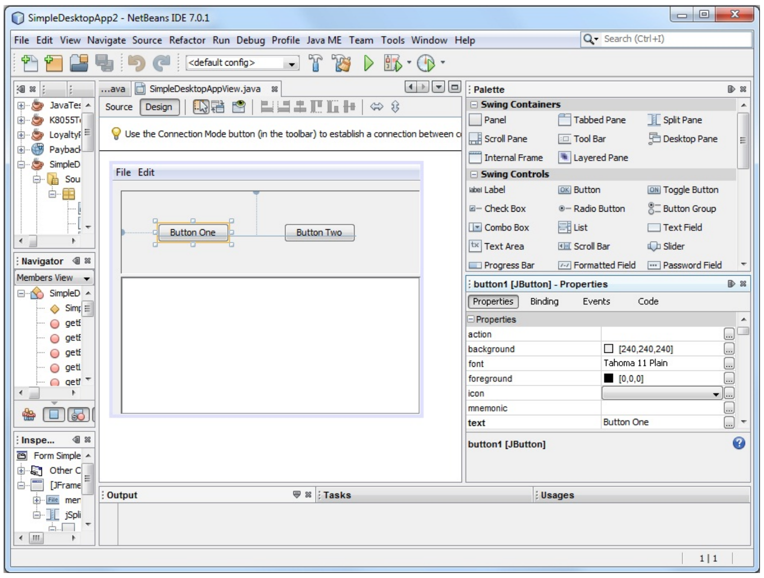


Fig. 3.8.: Two variants of modeling GUI logical structure in UML class diagram

It is to note that the logical structure modeled in this manner is not that helpful to embody the final structure illustrated in Figure 3.7. With increasing number of involved GUI elements and complexity of the internal structure, class diagram like this will reduce its readability dramatically. An alternative to cope with increasing complexity of GUI structure is to model all the involved GUI elements as normal properties instead of association-ends. This solution is shown on the right hand side of Figure 3.8. However, modeling in this way does not even show the basic tree structure achieved in the former model. Thus, the conclusion is that modeling logical structure only using standard UML class diagram is not sufficient, other supplement or enhancement is necessary.

The *visual layout* means the visual presentation of the GUI by spatially arranging GUI elements like buttons, input fields, labels and so on. Typical parameters defining visual layout are position, size, icon, background- and foreground color of each GUI element involved in the actual user interface. To configure these parameters, two methods are feasible: programming them directly in target language and manipulating them visually with the help of a software component classified as *GUI-Wizard*.

Programming GUI elements in an object-oriented target language means instantiating them and setting their properties appropriately. Mainstream GUI frameworks used to create the demo-example in this section are *composite pattern* [Gam+95] [FF04] based, so that there are clear parent-children relationships between involved elements. In this way, both the logical structure and visual layout can be established in a programmable manner. It is easy to understand that geometrical layout information like position and size are difficult to obtain when programming them. To solve this issue, some kind of intelligence, with which position and size information can be inferred, is achieved by using a technique called *layout manager*. Layout manager comes always with *container* together. For example, in order to line up several buttons in a container, in Swing, a container configured with *FlowLayout* can be used, while in Qt *QBoxLayout* and in WPF *WrapPanel* can be used respectively.


SimpleDesktopApp2 - NetBeans IDE 7.0.1
File Edit View Navigate Source Refactor Run Debug Profile Java ME Team Tools Window Help
Search (Ctrl+I)
<default config>
SimpleDesktopAppView.java
Source Design
File Edit
Button One
Button Two
Palette
Swing Containers
Panel Tabbed Pane Split Pane
Scroll Pane Tool Bar Desktop Pane
Internal Frame Layered Pane
Swing Controls
Label Button Toggle Button
Check Box Radio Button Button Group
Combo Box List Text Field
Text Area Scroll Bar Slider
Progress Bar Formatted Field Password Field
Navigator
Members View
Inspe...
Output
Tasks
Usages

Use the Connection Mode button (in the toolbar) to establish a connection between c

button1 [JButton] - Properties

Properties Binding Events Code

Properties	
action	
background	[240,240,240]
font	Tahoma 11 Plain
foreground	[0,0,0]
icon	
mnemonic	
text	Button One

button1 [JButton]

Fig. 3.9.: The Swing Designer shipped with NetBeans IDE

Developing GUI in source code is not productive, because the process is not intuitive. A better way is to use the GUI wizard, which is usually integrated in the IDE and supports visual manipulation of GUI elements. For each of the three GUI libraries used to create our example, there is a corresponding GUI wizard. Figure 3.9 shows the *Swing Designer* shipped with NetBeans IDE. All of them support GUI developers in a similar way. For a top-level container, e.g., a window or a dialog, there is always a *design view* associating with it. GUI developer chooses the required GUI elements from a Toolbox, which contains all the supported GUI elements in this context, positions them on the view, and edits them visually.

To reflect the manipulation of both the logical structure and visual layout, an XML-based *internal representation* is usually used by a GUI wizard. In WPF, this XML-representation is standardized as an declarative language, which can even be directly used by GUI developers to define the GUI. This language is called *eXtensible Application Markup Language* (XAML) [Mac12]. In other GUI libraries, such an XML representation is only used by the tool as an internal data structure.

The visual manipulation of GUI elements via GUI wizard is in fact only the efficient alternative to produce source code, which programs the GUI structure and layout in a target language. Thus, on all the three OOPL-platforms, the internal XML-representation is transformed into corresponding source code as the final *representation in target language* by GUI wizard automatically.

Up to this point, it should be clear that the both processes, in which visual layout of GUI can be developed in a target OOPL, are very difficult and even impossible to be modeled using UML. Because there is neither appropriate diagram types tailored for GUI layout modeling, nor a standard DSL shipped with UML to specify the GUI layout, just as OCL for specifying primitive and collection types.

Taking into account the analysis above, a solution to model GUI in the phase of creating the PIM of a GUI-based application is proposed, based on the following achievements:

- A platform independent GUI toolkit integrated into the MOCCA DPM
- A *Smart GUI Editor* integrated into UML 2 Designer, which supports visual manipulation of the GUI elements adopted in the GUI toolkit
- XOCL expressions used to represent the visual manipulation of GUI elements in the Smart GUI Editor

The *platform independent GUI toolkit* is created by reworking and refining the idea of modeling GUI in class diagram illustrated in Figure 3.8. Common GUI elements, which are ubiquitous on almost all the target platforms, are selectively adopted into this toolkit. Figure 3.10 summarizes all the important elements in the current GUI toolkit. The usage of these GUI elements is easy to understand by reading the class diagram, because they are just a high-level abstraction and compact restructure of the familiar concepts. To make difference between classes modeled by software developers in design model and classes belonging to MOCCA DPM, the latter are modeled always as UML *Data Type* stereotyped by *«DesignType»* additionally.

To be compatible with OCL, the abstract GUI element *Control*, which is the root of all the others, is derived from *OclAny* directly. Other abstract elements are introduced to classify GUI elements into different functional categories:

- *Control* represents a graphical entity of one kind or another that can be displayed on the screen. As the base type for all the other concrete GUI elements, properties and operations that are common for all GUI elements are defined in this type. Similar to concrete GUI libraries coming with OOPLs, significant properties and their semantics can be aware by reading the class diagram in Figure 3.10. It is to note that besides common properties and operations important GUI events are also declared in this base type by using platform independent event types, which belong to MOCCA DPM as well, and will be described in next section in detail.

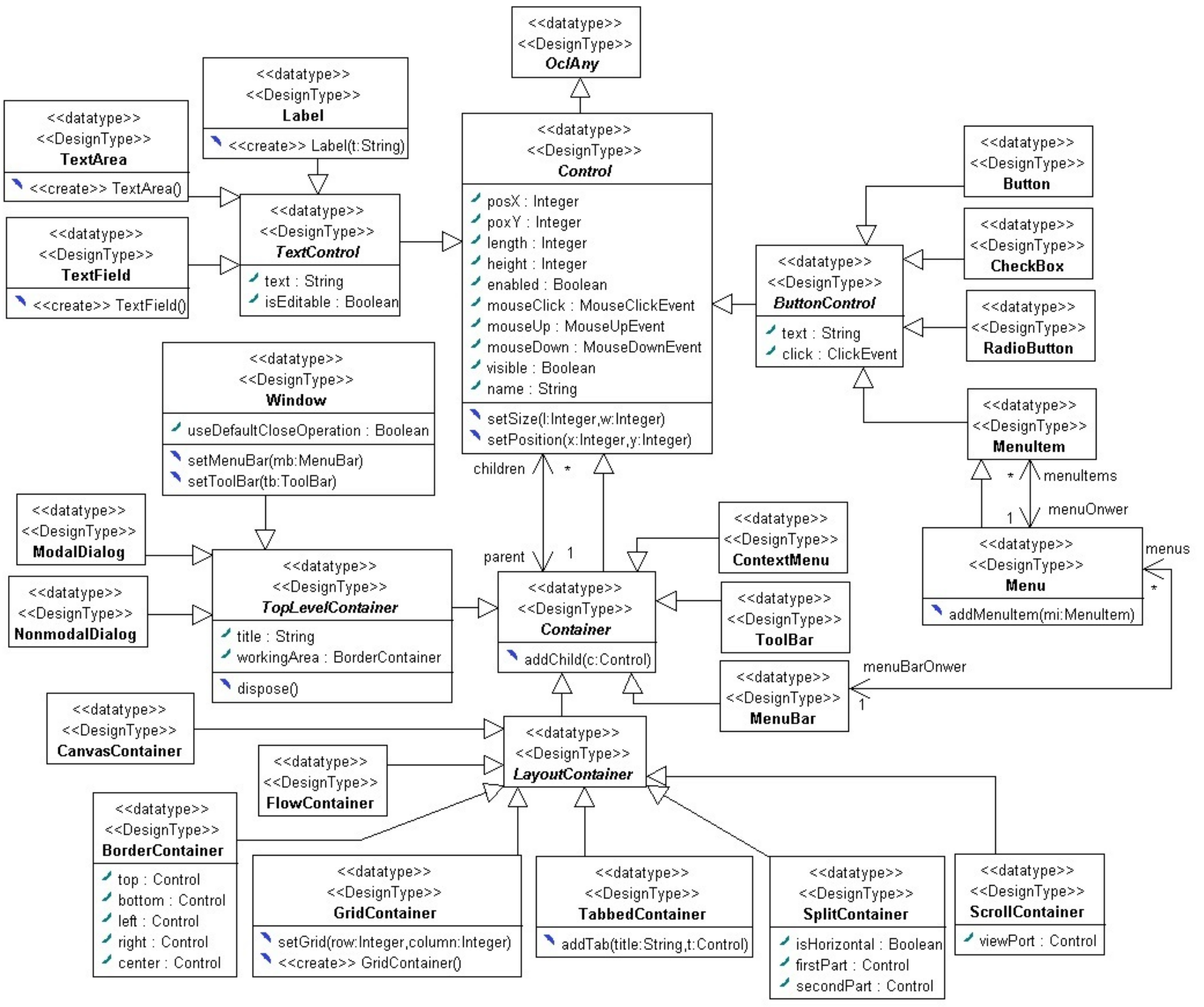

<<datatype>> <<DesignType>> OclAny
<<datatype>> <<DesignType>> Label
<<create>> Label(t:String)
<<datatype>> <<DesignType>> TextArea
<<create>> TextArea()
<<datatype>> <<DesignType>> TextField
<<create>> TextField()
<<datatype>> <<DesignType>> TextControl
text : String
isEditable : Boolean
<<datatype>> <<DesignType>> Control
posX : Integer
poxY : Integer
length : Integer
height : Integer
enabled : Boolean
mouseClick : MouseClickEvent
mouseUp : MouseUpEvent
mouseDown : MouseDownEvent
visible : Boolean
name : String
setSize(l:Integer,w:Integer)
setPosition(x:Integer,y:Integer)
<<datatype>> <<DesignType>> Button
<<datatype>> <<DesignType>> CheckBox
<<datatype>> <<DesignType>> RadioButton
<<datatype>> <<DesignType>> ButtonControl
text : String
click : ClickEvent
<<datatype>> <<DesignType>> MenuItem
<<datatype>> <<DesignType>> Window
useDefaultCloseOperation : Boolean
setMenuBar(mb:MenuBar)
setToolBar(tb:ToolBar)
children
*
parent
1
<<datatype>> <<DesignType>> ModalDialog
<<datatype>> <<DesignType>> NonmodalDialog
<<datatype>> <<DesignType>> TopLevelContainer
title : String
workingArea : BorderContainer
dispose()
<<datatype>> <<DesignType>> Container
addChild(c:Control)
<<datatype>> <<DesignType>> ContextMenu
<<datatype>> <<DesignType>> ToolBar
<<datatype>> <<DesignType>> MenuBar
menuBarOnwer
1
*
menuItems
1
menuOnwer
menus
*
<<datatype>> <<DesignType>> Menu
addMenuItem(mi:MenuItem)
<<datatype>> <<DesignType>> CanvasContainer
<<datatype>> <<DesignType>> FlowContainer
<<datatype>> <<DesignType>> LayoutContainer
<<datatype>> <<DesignType>> BorderContainer
top : Control
bottom : Control
left : Control
right : Control
center : Control
<<datatype>> <<DesignType>> GridContainer
setGrid(row:Integer,column:Integer)
<<create>> GridContainer()
<<datatype>> <<DesignType>> TabbedContainer
addTab(title:String,t:Control)
<<datatype>> <<DesignType>> SplitContainer
isHorizontal : Boolean
firstPart : Control
secondPart : Control
<<datatype>> <<DesignType>> ScrollContainer
viewPort : Control

Fig. 3.10.: Class hierarchy of the MOCCA DPM GUI elements

- *ButtonControl* represents an interactive element that can be labeled with a text and generates an *click event*.
- *TextControl* abstracts concrete GUI elements in respect of showing as well as editing text.
- *Container* is a control that can contain other controls. As shown in class diagram, the inheritance between *Container* and *Control* as well as the bidirectional association between them conform to the composition design pattern. Thus, the containers and their owned controls can be used in a nested manner.
- We distinguish between *TopLevelContainer* and *LayoutContainer*. The former has usually its own frame and title. Further more, their life cycle should be ended under some control. That is why the operation *dispose()* comes into play. The latter is used to arrange GUI elements with some kind of predictable layout semantics.

It is to note that all the properties (attributes and association-ends) defined in GUI elements have the *public* access modifier. So that they can be accessed in XOCL expressions directly. Two reasons make this decision reasonable: it is compatible with OCL standard and it is intuitive at the time of modeling. A detailed description for every GUI element in Figure 3.10 could be dispensable, because most of them can be considered just as a platform independent wrapper and preserve the familiar semantics and usages. Thus, only the concrete layout container will be addressed in detail.

- *CanvasContainer* has no intelligent predefined layout semantics to arrange GUI elements involved within it. It is meaningful when this layout container is used together with the Smart GUI Editor, with which geometrical information like position, size, etc., can be obtained visually so that the involved GUI elements are arranged according to the user's will. The *CanvasContainer* is similar to the layout container class *Canvas* of WPF and *JPanel* container class of Swing, with layout manager set to *null*.
- *FlowContainer* lines up GUI elements within it. If there is on more place to accommodate GUI elements in one line, the next line will be used. This container type abstracts e.g., the *WrapPanel* of WPF and container configured with *FlowLayout* in Swing.
- *BorderContainer* arranges and resizes its involving elements to fit in five regions: top, right, left, bottom and center by assigning the corresponding element to one of the five properties. This container preserve the semantics of *DockPanel* of WPF and container configured with *BorderLayout* in Swing.
- *GridContainer* manages GUI elements in a tabular way. The total number of rows and columns can be configured by invoking the *setGrid()* operation. This container type can be considered as the DPM counterpart of *Grid* of WPM and container configured with *GridLayout* in Swing.
- *TabbedContainer* manages GUI elements by grouping them into a tab with a given title. The *TabControl* of WPF and *JTabbedPane* in Swing can be considered as concrete example of this GUI container on different target platforms.
- *SplitContainer* divides its area into two parts, either horizontally or vertically based on the boolean property *isHorizontal*. The both GUI elements under its control are assigned to the properties *firstPart* and *secondPart* respectively.
- *ScrollContainer* provides an additional scrollable view of the element under its control. Concrete examples are *ScrollViewer* of WPF and *JScrollPane* in Swing.

Up to this point a well defined infrastructure in the form of MOCCA DPM GUI toolkit has been built, atop which the GUI structure can be modeled using these predefined model elements in class diagram. However, based on the considerations relating to Figure 3.8, without appropriate layout information only the structure modeled in class diagram is far from satisfaction. Inspired by the GUI designers shipped with modern IDEs, we suggest modeling GUI structure in class diagram as shown on the right hand side of Figure 3.8. That means, only the life cycle relationships between top level container (usually a window or a dialog) and its contained elements will be modeled. The internal tree structure and the precise layout of these GUI elements are manipulated visually within an additional software component called *Smart GUI Editor*.

The first version of the Smart GUI Editor was developed in the bachelor thesis of Mr. Döring [Dör10] as an add-on component of our UML 2 Designer. Figure 3.11 shows the main editor area of the Smart GUI Editor. Compared to popular GUI designer integrated into IDE, the GUI modeling is not driven by the Smart GUI Editor entirely. As aforementioned, a UML class representing a top level container together with all its involved GUI elements as normal properties must be created firstly. With this UML class as context, the Smart GUI Editor can be started and meanwhile a *view* is associated to it by making all the GUI elements available,

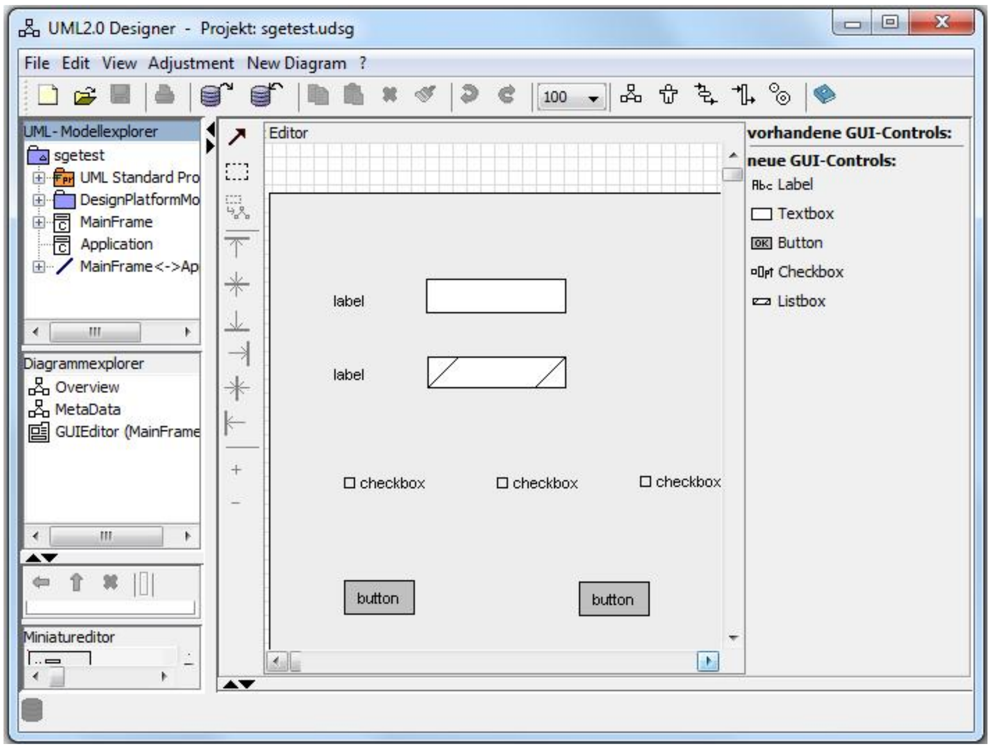


Fig. 3.11.: The Smart GUI Editor used to visually manipulate the GUI elements modeled in UML class diagram

which have been modeled as properties of this class, on the right hand side of the current view. From that moment on, all these GUI elements can be manipulated visually, such that both structure and layout can be established. This strategy is efficient and straightforward according to our requirement. Because the top level container class and its involved elements are modeled as normal. After that it should not be restructured by e.g., adding and removing contained properties, etc. All the visual manipulations in Smart GUI Editor do not change the underlying UML class structure, but are saved as a part of a design model together. Based on the following design requirements a solution is found to represent the visual manipulation of GUI elements in Smart GUI Editor.

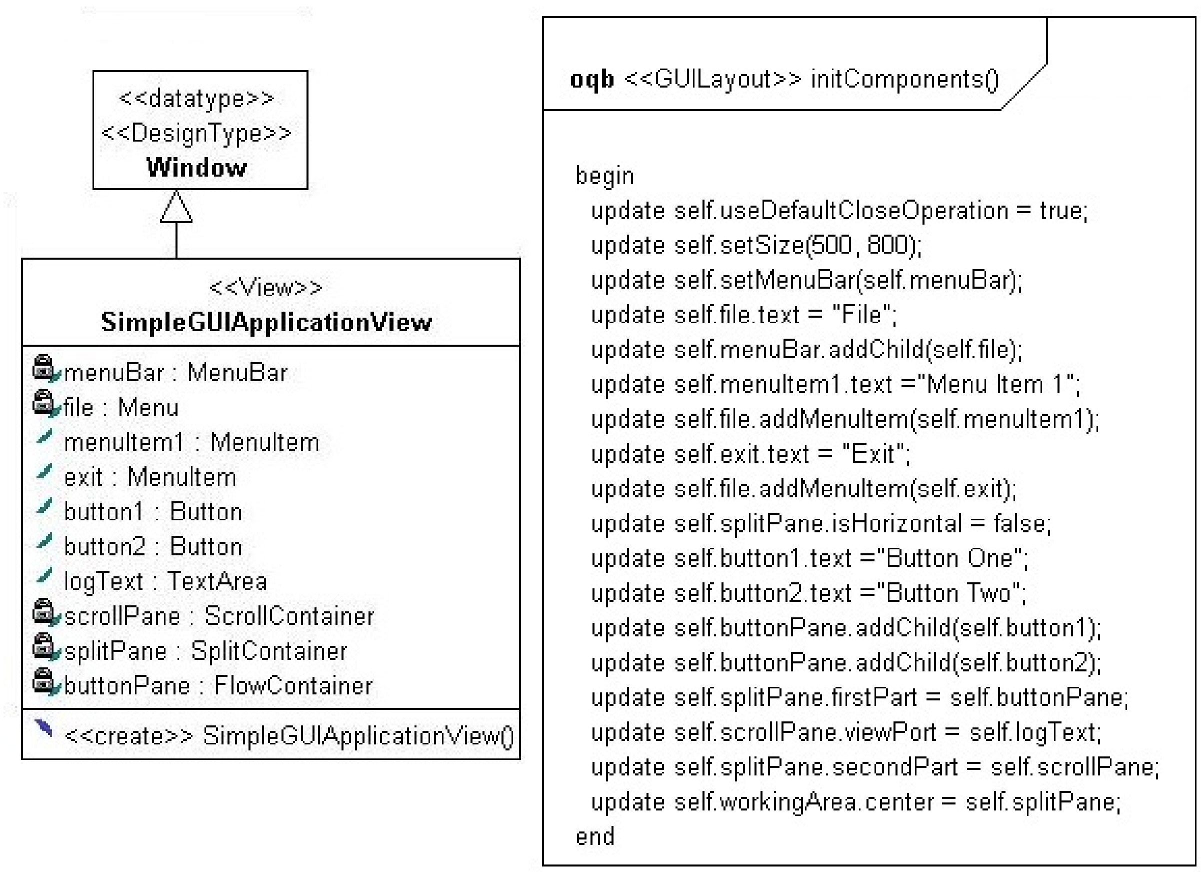
1. This intermediate representation must be platform independent, because this representation is a part of a design model, which is the PIM according to MDA terminology.

2. It should be possible and comfortable to map this intermediate representation onto different target platforms in the phase of model transformation.

3. At the time of writing this dissertation, the functionality of our Smart GUI Editor is still limited, so that sophisticated GUI cannot be established solely by using Smart GUI Editor. However, due to involving important layout containers, it should also be possible to specify the GUI layout via the intermediate representation manually. That means, the intermediate representation can also be understood and grasped by software modeler, although the representation should be generated by the Smart GUI Editor as output automatically.

Taking into account all the requirements above, the XOCL expressions are used in this work as the intermediate representation to reflect the visual manipulation done within our Smart GUI Editor on the one hand, on the other hand the expressions can also be edited by UML modeler to construct complex GUI at the time when the Smart GUI Editor is not powerful enough.

The extended language constructs, which upgrade the OCL into XOCL, add only the missing semantics, but do not affect the original OCL syntax. Thus, in this work, the XOCL imperative expressions are used as intermediate representation for GUI structure and GUI layout. The XOCL extension will be addressed in Section 3.4 in detail.



```
begin
  update self.useDefaultCloseOperation = true;
  update self.setSize(500, 800);
  update self.setMenuBar(self.menuBar);
  update self.file.text = "File";
  update self.menuBar.addChild(self.file);
  update self.menuItem1.text ="Menu Item 1";
  update self.file.addMenuItem(self.menuItem1);
  update self.exit.text = "Exit";
  update self.file.addMenuItem(self.exit);
  update self.splitPane.isHorizontal = false;
  update self.button1.text ="Button One";
  update self.button2.text ="Button Two";
  update self.buttonPane.addChild(self.button1);
  update self.buttonPane.addChild(self.button2);
  update self.splitPane.firstPart = self.buttonPane;
  update self.scrollPane.viewPort = self.logText;
  update self.splitPane.secondPart = self.scrollPane;
  update self.workingArea.center = self.splitPane;
end
```

Fig. 3.12.: Part one of the PIM for the GUI application shown in Figure 3.7: GUI structure and layout

Put all together, the simple GUI application introduced at the beginning of this section has been modeled. Even for a simple GUI application like this, some reasonable design decision can be made to organize an application in a flexible and maintainable manner, e.g., based on *Model-View-Controller* (MVC) [Sch09] design pattern. For the time being, we concentrate on structure and layout modeling for GUI using technique addressed in this section.

As suggested, the class on the left hand side of Figure 3.12 models the life cycle relationships between the top level window derived from DPM GUI element *Window* and its contained GUI elements. According to MOCCA, properties modeled as *private* have on *getter* and *setter* generated for them. Further more, *public* properties constrained by *readOnly* have either no *setter* generated. Thus, GUI elements that are not accessible from outside of the containing class should be modeled as *private* properties. The stereotype *«View»* marks the actual class as a presentation view according to MVC design pattern. Design pattern based stereotypes will be discussed in Section 3.3.4 thoroughly.

After creating this view class, the Smart GUI Editor can be started with the current view class as context to manipulate the involved GUI elements visually. According to our research goal, the XOCL expressions embodying visual manipulations in Smart GUI Editor will be generated and saved into a dedicated *OpaqueBehavior* attached to the view class directly. Further more, this *OpaqueBehavior* must be marked with stereotype *«GUILayout»*. On the right hand side of

Figure 3.12, the XOCL expressions specifying GUI structure and layout are shown together with the specific opaque behavior. For now, the XOCL expressions are written by software modeler manually.

3.3.3. Modeling Event Handling with XOCL Event Expression

In the previous section modeling GUI structure and layout in PIM have been addressed, which cover the first four aspects depicted in Figure 3.6. In this section modeling *event handling* in PIM will be discussed, which is the last issue left in Figure 3.6.

Modeling event handling in PIM is inherently challenging. There are diverse event handling mechanisms in different OOPLs. This diversity has to be hidden on the one hand. The ideal solution should be able to model event handling in a more intuitive and concise manner compared to programming event handling in target OOPL on the other hand. This objective can only be achieved by comparing typical event handling mechanisms in popular OOPLs to find out both the common aspects and the differences between them. This study has been done and described well in [LS10]. In this thesis, the important conclusions and the final solution are presented.

From the perspective of modeling, an event enables an object of a class (or a class itself) to publish changes of its state. Other objects and classes can then react to this change. This mechanism is usually called *Publishing - Subscription* model. Despite different implementations of this model in concrete target languages, the entire event handling process can be divided into four parts [KK07]:

1. *Static publishing* requires, that some kinds of events can be specified as members of their source. For example, events such as Button Click must be specified in GUI element representing a button. This part is only important for customized events. The most significant GUI events have been defined by GUI developers.

2. *Dynamic publishing* allows the transmission of the events. In both Java and C# this part is realized by a method, which triggers the execution of one or several dedicated event handling methods. Similar to static publishing, this part has been again implemented by GUI library designers.

3. *Static subscription* requires the implementation of all event-handling methods. In fact, exactly this part specifies what must be performed when an event occurs. This part has to be modeled by application modeler. Along with dynamic publishing, this part belongs to behavioral specification of an application model and should be modeled compactly using some kind of high-level action language, which will be addressed in Section 3.4.

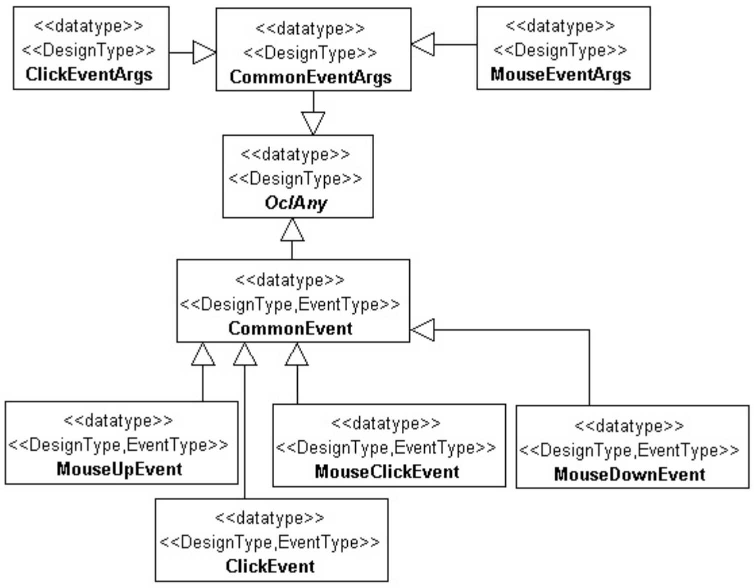
4. *Dynamic subscription* is done by establishing the connection between the event-source and the event handling method. Such a process is often called registration of event handlers. Exactly for this part of the entire event handling process, different target languages rely on diverse strategies. For example, various listener interfaces are used in Java to connect events with their handling methods loosely [HC08] while C# delegates set up these connections directly [Tro07].

To hide the platform specific details exposed in dynamic subscription, this part of event handling is further more decomposed into pieces to find out the most essential elements involved. Based on our analysis, the following elements can be considered as atomic:

- the *event source* object, which are usually the GUI elements of a window or the window itself,

- different types of *event* of an event source,
- *event handling methods*, which are implemented in event handler classes, and
- a *connection operator* that allows to set up the connection of an event to its event handling method.

As solution to involve all the essential aspects of dynamic subscription into the PIM in a concise and intuitive manner, we suggest extending OCL by a new expression, which is labeled by the key word *event*. In such an *OCL-event-expression* the new registration operator "∼" is used to establish the connection between an event on the left hand side and an event-handling method on the right hand side of this operator. We defined that the new OCL-event-expression in the above form has the type *OclVoid* and consequently no value. Because the OCL event expression also belongs to XOCL extension, so its rigorous concrete and abstract syntax definition will be addressed in Section 3.4. For now, we concentrate on its concrete usage in PIM.

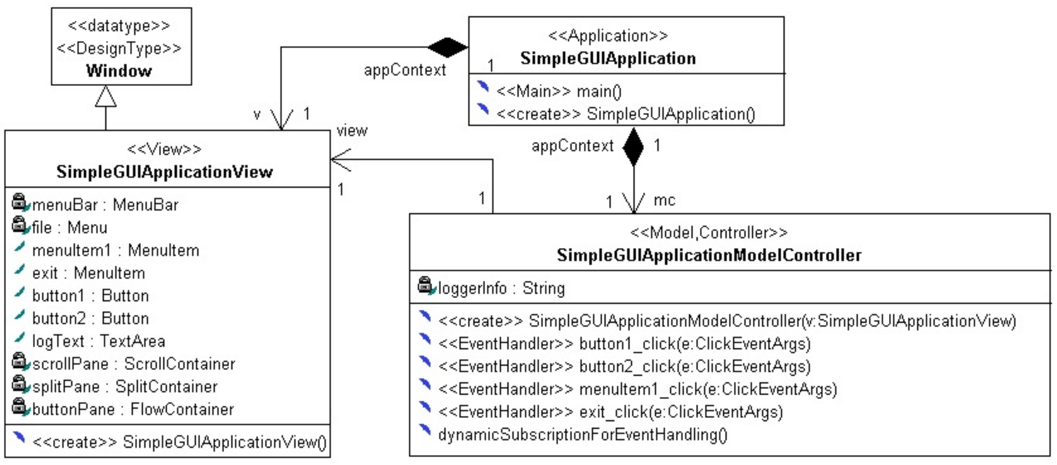


<<datatype>> <<DesignType>> ClickEventArgs
<<datatype>> <<DesignType>> CommonEventArgs
<<datatype>> <<DesignType>> MouseEventArgs
<<datatype>> <<DesignType>> OclAny
<<datatype>> <<DesignType,EventType>> CommonEvent
<<datatype>> <<DesignType,EventType>> MouseUpEvent
<<datatype>> <<DesignType,EventType>> MouseClickEvent
<<datatype>> <<DesignType,EventType>> MouseDownEvent
<<datatype>> <<DesignType,EventType>> ClickEvent

Fig. 3.13.: An excerpt of event related types adopted in the current MOCCA DPM GUI toolkit

Following the same principle with platform independent GUI elements involved in MOCCA DPM, event related types are also given in a sub-package of the superordinate GUI package. Figure 3.13 shows an excerpt of these types. With these event types defined, the GUI elements in DPM can be equipped with appropriate events. For example, in abstract type *ButtonControl* shown in Figure 3.10, the common event *click* with type *ClickEvent* is defined as a normal property. Taking into account the analysis in this section, it deals with static publishing of event handling for the platform independent GUI elements.

The design model in Figure 3.12 can be further completed as shown in Figure 3.14. In fact, this class diagram depicts the complete structure modeling in PIM of this GUI application based on MVC design pattern. For this simple application with only one string property recording user's actions, the *Model* and the *Controller* can be merged as a whole, that are expressed by stereotyping the design class *SimpleGUIApplicationModelController* with *«Model»* and *«Controller»* at the same time. For the time being, the attention should be paid to the operations stereotyped by *«EventHandler»* and the operation named *dynamicSubscriptionForEventHandling()*. The former are event handling operations that must be connected to appropriate GUI events.



<<datatype>>
<<DesignType>>
Window
<<Application>>
SimpleGUIApplication
<<Main>> main()
<<create>> SimpleGUIApplication()
appContext
v
1
view
1
1
appContext
1
1
mc
<<View>>
SimpleGUIApplicationView
menuBar : MenuBar
file : Menu
menuItem1 : MenuItem
exit : MenuItem
button1 : Button
button2 : Button
logText : TextArea
scrollPane : ScrollContainer
splitPane : SplitContainer
buttonPane : FlowContainer
<<create>> SimpleGUIApplicationView()
<<Model,Controller>>
SimpleGUIApplicationModelController
loggerInfo : String
<<create>> SimpleGUIApplicationModelController(v:SimpleGUIApplicationView)
<<EventHandler>> button1_click(e:ClickEventArgs)
<<EventHandler>> button2_click(e:ClickEventArgs)
<<EventHandler>> menuItem1_click(e:ClickEventArgs)
<<EventHandler>> exit_click(e:ClickEventArgs)
dynamicSubscriptionForEventHandling()

Fig. 3.14.: Part two of the PIM for the GUI application shown in Figure 3.7: event handling

The latter is the operation, whose method takes over the connecting between events and their handling operations by using XOCL-event-expressions shown in Listing 3.3. Along with class diagram in Figure 3.14, it is easy to get the idea that the first XOCL event expression registers the *button1_click* operation as the event handling operation of the *click*-event, which can be launched by *button1* property. The corresponding navigation to an event property as well as to event handling operation abides by classical OCL syntax and semantics [WK03] [OMG10a].

```
begin
  event: self.view.button1.click ~ self.button1_click ;
  event: self.view.button2.click ~ self.button2_click ;
  event: self.view.menuItem1.click ~ self.menuItem1_click ;
  event: self.view.exit.click ~ self.exit_click ;
end
```

Listing 3.3: XOCL event expressions specifying the dynamic subscription logic in the operation dynamicSubscriptionForEventHandling()

3.3.4. The Profile in Design Platform Model

As introduced in Section 2.1.3, stereotypes defined in UML profiles extend UML model elements, on which one or more stereotypes are attached, with extra semantics. This semantic enhancement helps creating compact PIMs by declaring required functions in PIM, which could be very complicated while being mapped into concrete OOPLs. For example, a normal UML class in PIM, which models a program partner in some system and contains several properties like name and description, can be stereotyped as *persistence* class, if its instances are to be saved in the long term. In fact both the *Annotation* in Java and *Attribute* in .Net languages support providing development objects with additional information in a non-programming manner, very similar to the stereotype mechanism in UML. In programming-language-centric systems like Java and .Net, corresponding compilers interpret annotations or attributes to generate required functions, whereas in UML modeling a model mapper can process stereotypes in an appropriate manner. For MOCCA, all the stereotypes, which can be used in the design model, are defined in the profile *DPMProfile* of MOCCA DPM. Based on their characteristics, these stereotypes are classified into three categories.

Design pattern based stereotypes

The *design pattern based stereotypes* are used to decorate a UML model element with design pattern related information. A design pattern describes a general solution to a design problem that recurs in various projects. Design patterns are usually formulated using UML [FF04]. However, UML does not keep track of pattern-related information when a design pattern is applied. Thus, it is hard for a designer to identify design patterns in software system design. Using UML profile and stereotypes to involve design pattern information into UML model is not novel in this work. A general purpose profile containing stereotypes that support working with object-oriented design pattern was discussed well in [DY03]. We call the stereotypes defined in [DY03] *general purpose*, because, to some extent, they can be considered as meta-level stereotypes to define, but not to declare, which design pattern related information is in use. For example, instead of providing a *«Singleton»* stereotype directly, the meta-level stereotype *«PatternClass»* with its tagged value *pattern*, to which the design pattern name *Singleton* is assigned, can be used to mark a class as singleton class. Different from this strategy, the stereotypes defined in this work are concrete and can be used directly to communicate pattern-related information.

According to the current status of MOCCA, the following design patterns are supported directly by the corresponding stereotypes. Of course, MOCCA is not restricted to support only these design patterns, other mature patterns as well as even newly designed patterns can be added into this group with corresponding mapping rules as prerequisite.

- Three-Layer-Architecture Pattern
- Model-View-Controller Pattern
- Data-Mapper Pattern
- Singleton Pattern

Three-Layer-Architecture is a design pattern that is used primarily to develop enterprise-level distributed applications. The three layers involved are presentation layer, business objects layer (also called domain logic layer, application layer etc.) and data persistence layer. According to MOCCA modeling paradigm, layers are modeled using UML *package*. Thus, stereotypes with respect to this design pattern can be applied only to UML package.

- *Presentation layer* is about how to handle the user interaction with the system and how to display operation result to the users [Fow+02]. Stereotype *«PresentationLayer»* is used to mark a package as the presentation layer. After that, all the child-elements defined in this package belong to the presentation layer semantically.
- *Business objects layer* implements the work that this application actually needs to to. The outermost package of this layer is stereotyped by *«BusinessObjectsLayer»*.
- *Data persistence layer* serves as interface, through which important data objects coming with application domain can be saved in and retrieved from diverse data storages, such as files with different formats or data base systems. Stereotype *«PersistenceLayer»* marks the outermost package of this layer.

Figure 3.15 illustrates a classical three layer architecture, in which a single layer knows about or, in other words, accesses only the layer directly beneath it. This figure shows also the typical usages of the stereotypes explained before.

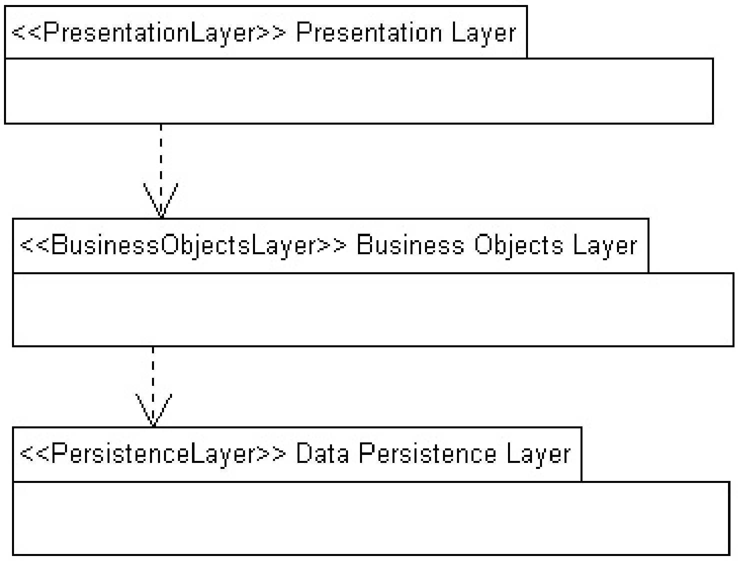

<<PresentationLayer>> Presentation Layer
<<BusinessObjectsLayer>> Business Objects Layer
<<PersistenceLayer>> Data Persistence Layer

Fig. 3.15.: Package diagram illustrating three layer architecture

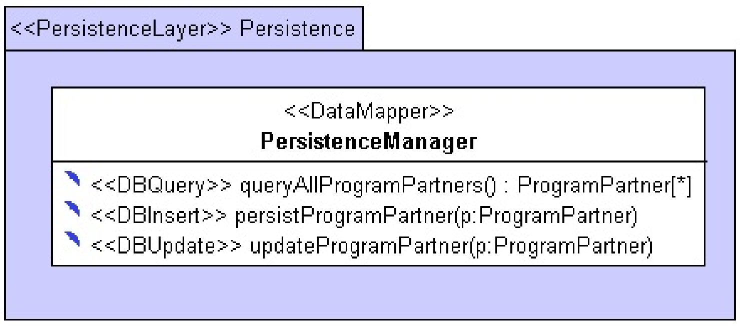
Model-View-Controller is a well-known design pattern that goes back to one of the most important principle of object-oriented design - separate of concerns, that means, don't make one object responsible for too much [HC08]. MVC has been used to design GUI library, e.g., Java Swing, on the one hand, on the other hand can be used to organize a GUI based application very well. As described in [Gam+95], *Model-View-Controller* consists of three kinds of objects:

- *Model* is the application object, which manages or represents the entire data structure of the application domain. Model can be complicated and consist of many classes. According to MOCCA, stereotype *«Model»* is applied to the outermost wrapper class of the entire domain model.
- *View* is the screen presentation of the domain model. There can be different views based on one domain model, representing the domain data from different perspectives. In MOCCA stereotype *«View»* is used to mark such one or several view-classes.
- *Controller* defines the way the user interface reacts to user input. Simply speaking, the controller-class consists of event handling operations to deal with GUI-events. In MOCCA stereotype *«Controller»* is used to mark a class for this purpose.

The example of Figure 3.14, which was used in the previous section to depict modeling GUI layout and event handling, shows also a typical usage of MVC-related stereotypes defined here. As explained, in this example, the data model is so simple that it can be merged into controller class.

Data-Mapper is a design pattern solution to cope with object-relational mapping [Fow+02] [Sch09]. Objects and relational databases have different mechanisms for structuring data. When data must be transferred between the two schemes, some kind of conversion is necessary. To implement such a conversion, ideally, the in-memory objects representing business domain should know nothing about the underlying data base structure. If not so, changes in one tend to ripple to the other.

According to [Fow+02], *Data-Mapper* is defined as a software component that separates the in-memory objects from the database. Its responsibility is to transfer data between the two and also to isolate them from each other. With Data Mapper the in-memory objects do not need to know even that there's a database present; they need no SQL interface code, and certainly no knowledge of the database schema. Since it's a form of mapper, Data-Mapper itself is even unknown to the business domain objects. In MOCCA, the stereotype *«DataMapper»* is defined to mark a class as the data mapper component. Figure 3.16 shows a typical use case. The stereotypes applied to operations defined in data mapper class will be addressed later in this section.



<<PersistenceLayer>> Persistence
<<DataMapper>>
PersistenceManager
<<DBQuery>> queryAllProgramPartners() : ProgramPartner[*]
<<DBInsert>> persistProgramPartner(p:ProgramPartner)
<<DBUpdate>> updateProgramPartner(p:ProgramPartner)

Fig. 3.16.: Data-Mapper class resides in the persistence layer

A concrete implementation of data mapper class in a target language can be very complicated. However, as mentioned in the second application scenario of Section 2.3.1, O/R mapping as a common issue has been taken into account in many popular OOPL-platforms, among which, the JPA in JEE is directly based on Data-Mapper pattern, other platform like SAP ABAP integrates data base access into the language architecture seamlessly, so that mapping a PIM-data-mapper into PSM-data-mapper by the corresponding model mapper is feasible.

Singleton Pattern ensures a class has only one instance, and provides a global point of access to it [FF04]. There are many singleton objects in practice: thread pools, caches, dialog boxes, objects that handle preferences and registry settings, objects used for logging, and objects that act as device drivers. As the definition indicates, the *«Singleton»* stereotype is applied to a design class directly.

Common function based stereotypes

The *Common function based stereotypes* are used to enhance model elements in PIM with additional semantics, which are ubiquitous on diverse implementation platforms but represented platform specifically. According to the current status of MOCCA, the following stereotypes are defined and fall into this category. Analogous to design pattern based stereotypes, more of them could be added in the later development.

- *«Main»* can only be applied to an operation in PIM. Such an operation is considered as *execution entry* to the entire application modeled. On different platforms, varying requirements could be given for this operation. For example, standalone applications implemented in diverse target OOPLs are usually equiped with a *main()* method, of course, with varying signatures in detail, whereas container-controlled application (components) is started by invoking some kind of *life cycle callback* operation.

- *«EventHandler»* marks an operation as event handling method, which implements the logic for the static subscription part of the entire event handling process. As depicted in Section 3.3.3, the XOCL event expressions do not require complete operation signature on the right hand side of the connection operator. Because in the most situations, event handling operations have predefined parameters. By using this stereotype the event handling operations can be highlighted on the one hand, on the other hand this stereotype helps the model mapping by providing useful hints for a special group of operations.

- *«GUILayout»* is added to special *opaque behaviors*, which must be attached to view-classes stereotyped by *«View»*, to record GUI-layout information.
- *«Application»* is applied to a design class, which is considered as the startup component of the entire system. For example, the singleton session bean [LB10], which should be initialized eagerly by annotating it with *@Startup* can be modeled in PIM with this stereotype, whereas on JSE platform, a normal Java class containing *main()* [1] operation can be modeled as *Application*-class.

Figure 3.14 in the previous section shows already an example with typical usages of all the stereotypes addressed in this section. It is to note that the UML standard stereotype *«create»* [RQZ07] marking an operation as constructor is also used in the the MOCCA modeling framework, and can be considered as a stereotype falling into this group. In fact, to specify a common aspect in PIM, the standard UML stereotypes should be taken into account firstly. If no matching found, a new one will be defined.

Application domain based stereotypes

As the name suggests, this group of stereotypes is designed to simplify modeling software applications targeted to special domains. This strategy is generally feasible, because for each application domain, domain specific software have their own characteristics, most of which have been studied and abstracted well. Platform independent but domain specific stereotypes can be used intuitively to give model elements domain specific semantics, which can be mapped onto concrete programming language platforms in the form of boilerplate code. As mentioned in Section 2.4, just following the idea above, domain specific stereotypes for modeling SOA based applications [HLT03] [Lóp+07], for modeling context-aware web application [Kap+09], etc. have been developed in the frame of MDA.

The stereotypes to be addressed in this sections are conceived to model three layered, enterprise applications primarily. In this field, functions like persisting data in relational databases, concurrency management, role-based security are almost indispensable for applications to be developed. Inspired by OMG EDOC [OMG04], the following stereotypes are involved into the current *DPMProfile*:

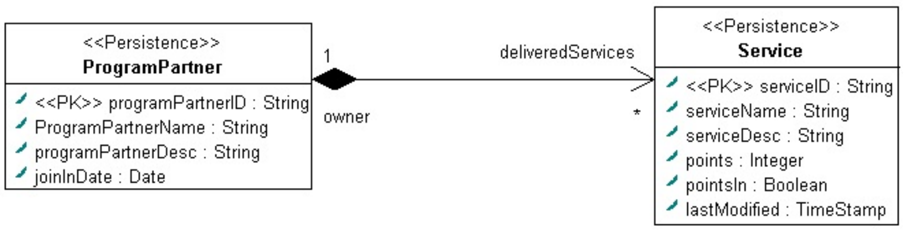


Fig. 3.17.: Domain logic classes, whose instances are to be saved into RDBMS

- *«Persistence»* together with *«PK»* are used to model a class, whose instances are to be saved into relational database. The former is applied to a design class, the latter denotes

[1] This main-operation is meant to the actual execution entrance. In Java every class can contain a main operation, but only one can be set to the startup operation for the entire application. According to the modeling semantics, whatever this operation is called, it must be constrained by *«Main»* stereotype.

one or more properties as *primary key* according to relational data model. Figure 3.17 shows an example with two such persistence classes.

- *«DBInsert»*, *«DBQuery»* as well as *«DBUpdate»* denote operations with exactly the same semantics as these stereotypes indicate. Usually, database-related operations reside in the data mapper class highlighted by *«DataMapper»* as addressed before. The example of a data mapper class in Figure 3.16 depicts also the typical usages of these three stereotypes.
- *«BusinessRole»*, *«AdminRole»* and *«CommonRole»* are used to mark design classes, which represent the potential system users (also called system actors). Most enterprise applications are designed to serve a large number of clients, and users are not necessarily equal in terms of their access rights. An administrator might require hooks into the configuration of the system, whereas unknown guests may be allowed a read-only view of data [LB10]. Due to this fact, potential users of the application are grouped into categories with defined *roles*, access can be then allowed or restricted to the role itself during system configuration. As name indicates, *«AdminRole»* denotes a class, whose operations can be executed by administrator role of the system. There can be only one class marked by this stereotype. The *«CommonRole»* indicates, some of the operations defined in the actual class are general purpose query operations be called by even non-registered guests. *«BusinessRole»* marks a system actor, who can execute some kind of business operations after login into the system. There can be different business roles involved in a system.
- *«BusinessOperation»* and *«CommonOperation»* stereotype *transaction*-operations that can be invoked concurrently by different system actors, who are now using the system. The logic, especially, the non-query logic in transaction operations must be controlled in terms of transaction management. Common operations are defined into a class, which has been stereotyped at least by *«CommonRole»*, whereas business operations can exist either in business role class or administrator role class.

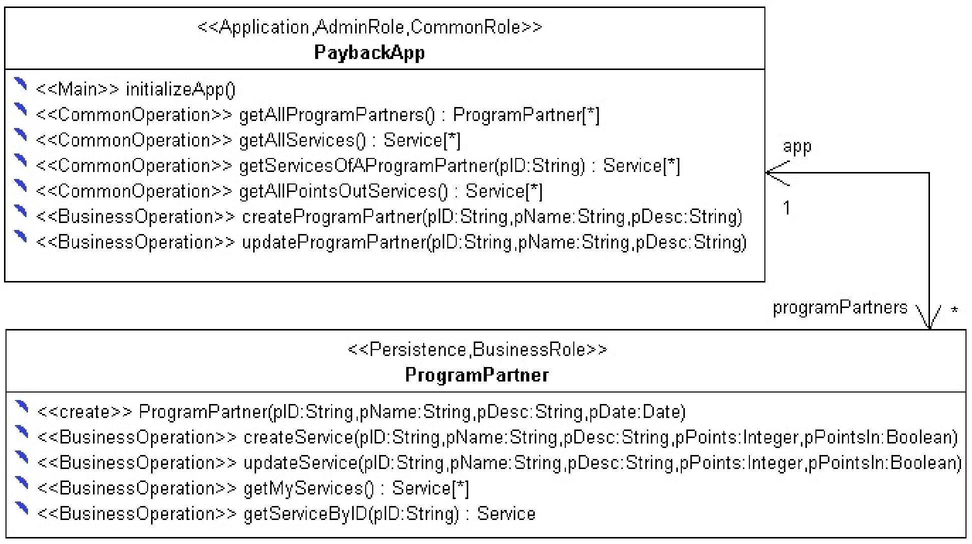

<<Application,AdminRole,CommonRole>>
PaybackApp
<<Main>> initializeApp()
<<CommonOperation>> getAllProgramPartners() : ProgramPartner[*]
<<CommonOperation>> getAllServices() : Service[*]
<<CommonOperation>> getServicesOfAProgramPartner(pID:String) : Service[*]
<<CommonOperation>> getAllPointsOutServices() : Service[*]
<<BusinessOperation>> createProgramPartner(pID:String,pName:String,pDesc:String)
<<BusinessOperation>> updateProgramPartner(pID:String,pName:String,pDesc:String)
app
1
programPartners
*
<<Persistence,BusinessRole>>
ProgramPartner
<<create>> ProgramPartner(pID:String,pName:String,pDesc:String,pDate:Date)
<<BusinessOperation>> createService(pID:String,pName:String,pDesc:String,pPoints:Integer,pPointsIn:Boolean)
<<BusinessOperation>> updateService(pID:String,pName:String,pDesc:String,pPoints:Integer,pPointsIn:Boolean)
<<BusinessOperation>> getMyServices() : Service[*]
<<BusinessOperation>> getServiceByID(pID:String) : Service

Fig. 3.18.: Role-based classes with transaction operations

Putting all the role-based stereotypes into action, Figure 3.18 shows an example that is only a piece of a complex PIM to be discussed as experimental result in Section 5.2. For the time being, only the modeling semantics of the stereotypes aforementioned should be aware. Starting with design class *ProgramPartner*, it is a persistence class and meanwhile represents a business role. After submitting required information, which must be saved permanently, and login onto the system, a program partner can do something within the system, e.g., create a brand new service or update an existent one. Such tasks can be modeled as business operations. Navigating to the design class *Payback*, it is a start-up component, which is in charge of initializing the entire system by means of calling *initializeApp()* operation. This class models also a common role, whose common services in the form of common operations can be called by all potential roles concerned by the system. Further more, this class involves also business operations that can only be executed by system administrator. This modeling intention is indicated by adding *«AdminRole»* stereotype onto the class.

As mentioned at the beginning of this section, there are many well-defined profiles together with stereotypes for different application domains. MOCCA does not exclude working with the third party DPM stereotypes. With some effort of configuring and extending the existent compiler components, it is generally feasible to involve more and more profiles coping with different modeling problems for different application domains.

3.4. Modeling Behavior in XOCL

As introduced in Section 3.1, for the objective of compact and complete behavior modeling, the OCL has been extended with a small set of language constructs to become a full-fledged action language called *eXecutable OCL* (XOCL). The general support of using textual AL in UML and the important reasons of choosing OCL as the start point to design an AL have been already addressed in Section 2.1.4 and 2.2.2 respectively. To help understanding our considerations, the advantages of OCL are briefed again as follows:

- As a standard part of UML 2 Specification, OCL covers large part of the entire Action Semantics of UML. Only the semantics involving changing the states in the model and explicit control flows are missing. Such missing semantics can be easily added to the standard OCL with new syntax constructs.

- The OCL collection types and their predefined operations are powerful in terms of expressiveness and concise in terms of syntax. As depicted in Listing 2.2, together with OCL-body expression, very complex query operations in PIM can be specified both concisely and exactly.

Towards upgrading OCL to a real action language, three important issues must be addressed:

1. Identification of the action semantics, which are necessary for an action language but not yet involved in the OCL

2. Definition of concrete syntax to be added to OCL, which can express the identified action semantics as additional language constructs

3. Extension as well as modification of OCL abstract syntax to represent the language constructs brought in by new concrete syntax.

For the first issue, other researchers have worked out a generally accepted result, which were discussed in [HP04] and [JZM07] well. Moreover, our early research [LS11] on this issue was also based on these achievements. In this work, the important action semantics, which are either not involved or only partially covered by OCL, are summarized in Table 3.1 to Table 3.5.

Tab. 3.1.: Actions with respect to updating states of an object

Actions:	Brief description:
AddStructuralFeatureValueAction	According to UML terminology, the attributes and association-ends are structural features. This action is write action for adding values to them.
RemoveStructuralFeatureValueAction	This action is a write action that removes one value from the set of values contained in the specified structural feature.
ClearStructuralFeatureValueAction	This action is a write action that removes all values of a structural feature.

Tab. 3.2.: Actions with respect to updating local variables

Actions:	Brief description:
AddVariableValueAction	This is a write action for adding values to local variables.
RemoveVariableValueAction	This is a write action that removes one value from the set of possible variable values.
ClearVariableValueAction	This is a write action that removes all values of a local variable.

Tab. 3.3.: Actions used to create and destroy objects respectively

Actions:	Brief description:
CreateObjectAction	This action creates an object that conforms to a statically specified classifier and puts it on an output pin at runtime. Simply put, it instantiates an object of a class.
DestroyObjectAction	In contrast with create object action, this action destroys an object of a class.

Tab. 3.4.: Actions that are partially covered by OCL

Actions:	Brief description:
ReplyAction	This action returns the values to the caller of the previous call, completing execution of the call.
CallOperationAction	This action transmits an operation call request to the target object, where it may cause the invocation of associated behavior.

The second issue concerns designing textual representation for the action semantics identified in the previous step. In other words, how the surface language elements corresponding those action semantics look like. For this issue, different opinions are held by different researchers. Some of them consider that the syntactical appearance of OCL is strange, because it roots in no widely spread programming language, and this unfamiliar syntax makes it difficult for developers to roll in using OCL. Thus, a Java-like modification of OCL syntax was suggested [Rum02] [Rum11].

Based on our own research, we agree to the opinion that the OCL syntax is unfamiliar at the first glance, but not to that the original syntax has to be aligned with other programming language. Because in the spirit, OCL is a declarative modeling language that should be distinguished

Tab. 3.5.: Action semantics representing explicit control flows

Structured Activities:	Brief description:
SequenceNode	This is a structured activity node that executes its actions in order. In other words, it represents sequential execution of statements in programming language.
ConditionalNode	This is a structured activity node that represents an exclusive choice among some number of alternatives. In practice, it represents the *if-else* statement in programming language.
LoopNode	This is a structured activity node that represents a loop with setup, test against loop condition and execution body. In practice, it corresponds to diverse loop statement in programming language.

from imperative programming languages. Further more, OCL is defined and maintained by OMG as a de facto modeling language, we believe that more and more developers will get familiar to its concise syntax. Based on these considerations, the XOCL extension in this work abides by an important design philosophy: *the missing semantics for making OCL as AL should be added by concise and easy to understand concrete syntax elements that can be clearly distinguished from original OCL and do not modify the original OCL expressions.*

Following this philosophy, the grammar rules defining new XOCL language constructs are summarized in Listing 3.4. A complete list of XOCL grammar rules is given in Appendix A, whose OCL part is based on [OMG10a]. It is to note that all the grammar rules are given in the BNF notation, but in the form required by GOLD Parsing System [Coo13]. Because GOLD Parsing System is used in this thesis to implement an XOCL compiler. According to GOLD, non-terminals are delimited by the angle brackets and terminals are delimited by single quotes or not delimited at all. Up to this point, each of the grammar rules in Listing 3.4 can be explained in conjunction with its action semantics.

The XOCL *block-expression* is defined by the grammar rules from Line 1 to Line 2 in Listing 3.4. It provides the semantics of sequence node listed in Table 3.5. On the one hand, it can be used as a top level expression like OCL *body expression* but taking a *non-query-operation* as context, whereas behavior of a query operations can be specified by using OCL body expression. On the other hand, it can be used as parts in other control-flow-related compound expressions to be addressed later. In essence, it is the extension point, through which imperative semantics are added into OCL. According to the grammar rules, a block expression can be empty or enclose arbitrary number of other expressions. Allowed expressions are defined by grammar rules from Line 7 to Line 14. It is worth noting that compound expressions have their own end marker, whereas simple expressions are finished by a semicolon. Instead of using curly braces to denote the scope of block expressions, two keywords *begin* and *end* are brought into XOCL. Because, firstly, curly braces are used in OCL to define collection literals [WK03] and secondly, OCL uses keywords like *then* and *endif* to denote scope. Thus, we use the same style for XOCL extension.

The XOCL *while-expression* is defined by the grammar rule at Line 16. It provides the semantics of loop node listed in Table 3.5 and consequently, it models *explicit* loop semantics. We highlight "explicit", because the loop semantics is supported by OCL to some extent. The predefined OCL loop operations [WK03] enable to loop over the elements in a collection by taking each element in the collection and evaluating an expression on it. The *while* expression can be used as a generic loop independent of a collection, in contrast.

The grammar rule at Line 18 turns the OCL predefined *iterate()* operation into a generic *for each* loop [HC08] [Tro07] coming with modern OOPLs. The first part within the generic *iterate()* operation defines a *iterator* variable, whereas the second part is a block expression, in which any operations can be done for the iterator variable.

The XOCL *if-expression* is defined by the grammar rules from Line 20 to Line 22 in Listing 3.4. It provides the semantics of conditional node listed in table 3.5. The XOCL *if-expression* enhances the original OCL *if-expression* covered by the Line 22 with the ability to execute non-query logic in both the *then*-branch and the optional *else*-branch.

```
1  <BlockExpCS> ::= 'begin' <ImperativeExpListCS> 'end'
2                 | 'begin' 'end'
3
4  <ImperativeExpListCS> ::= <ImperativeExpListCS> <ImperativeExpCS>
5                          | <ImperativeExpCS>
6
7  <ImperativeExpCS> ::= <WhileExpCS>
8                      | <IfExpCS>
9                      | <OCLVarDeclarationCS>        ';'
10                     | <AssignExpCS>                ';'
11                     | <DestroyObjectExpCS>         ';'
12                     | <ReplyExpCS>                 ';'
13                     | <NonQueryFeatureCallExpCS>   ';'
14                     | <EventExpCS>                 ';'
15
16 <WhileExpCS> ::= 'while' <OCLExpCS> <BlockExpCS> 'endwhile'
17
18 <OCLIterateExpCS> ::= 'iterate' '(' <OCLVarDeclarationCS> '|' <BlockExpCS> ')'
19
20 <IfExpCS> ::= 'if' <OCLExpCS> 'then' <BlockExpCS>  'endif'
21             | 'if' <OCLExpCS> 'then' <BlockExpCS> 'else' <BlockExpCS> 'endif'
22             | 'if' <OCLExpCS> 'then' <OCLExpCS>   'else' <OCLExpCS>   'endif'
23
24 <AssignExpCS> ::= 'update' <OCLFeatureCallExpCS> '=' <OCLExpCS>
25                 | 'update' <OCLFeatureCallExpCS> '=' <CreateObjectExpCS>
26
27 <CreateObjectExpCS> ::= 'new' <OCLFullNameExpCS>
28
29 <DestroyObjectExpCS> ::= 'delete' <OCLFeatureCallExpCS>
30
31 <NonQueryFeatureCallExpCS> ::= 'update' <OCLFeatureCallExpCS>
32
33 <ReplyExpCS> ::= 'return' ID
34
35 <EventExpCS> ::= 'event' ':' <OCLFeatureCallExpCS> '~' <OCLFeatureCallExpCS>
```

Listing 3.4: Grammar rules in terms of extending OCL into XOCL

The XOCL *assignment-expression* is defined by the grammar rules from Line 24 to Line 25 in Listing 3.4. It supports all the write actions listed in Table 3.1 and Table 3.2. With this expression, the system states represented by properties and local variables can be modified and consequently, the XOCL can be considered as AL-ready. In modern programming languages, assignment is usually denoted by the "=" sign. However, in OCL "=" is used as *equal to* in logical expression or to define a local variable for the first time. Instead of bringing in a new symbol to express assignment, like done in [JZM07] and preserved in our early work [LS11], the keyword *update* is used to make "=" symbol context-sensitive. That means, if "=" is used as in programming language for assignment in an XOCL block expression, the *update* must be prefixed to give a hint for modification semantics, otherwise it is used as defined in original OCL.

The XOCL *create-object-expression* is defined by the grammar rule at Line 27 in Listing 3.4. It implements the create object action listed in Table 3.3. The *new* operator is used as in most OOPLs to create an object, after that, the object must be assigned to either property or variable.

The XOCL *destroy-object-expression* is defined by the grammar rule at Line 29 in Listing 3.4. It implements the destroy object action listed in Table 3.3. At the time of creating a PIM, the application logic cannot rely on the garbage collection system of the later target platform. Thus, the language construct of modeling destructing an object explicitly is provided by this expression with the *delete* operator.

The XOCL *non-query-operation-call-expression* is defined by the grammar rule at Line 31 in Listing 3.4. It completes the call operation action described in Table 3.4 with the ability to call non-query operations in XOCL block expressions. Because non-query operations can modify system states, the *update* keyword is used again to express this intention.

The XOCL *reply-expression* is defined by the grammar rule at Line 33 in Listing 3.4. It completes the reply action described in Table 3.4 with the ability to return the object to the caller explicitly. OCL body expression for specifying query operation implies to return an object or a collection of objects as the query result. To support the reply expression, the keyword *return*, whose semantics is identical to return statement in programming language, is brought in.

The XOCL *event-expression*, whose usage and contribution in the context of modeling GUI in PIM have been addressed in Section 3.3.3, is defined by the grammar rule at Line 35 in Listing 3.4.

```
 begin
  osToBeSorted: OrderedSet(Integer) = OrderedSet{5..1};
  indexOfInsertedElem : Integer = 2;

  while indexOfInsertedElem <= osToBeSorted->size()
    begin
      insertedElem : Integer = osToBeSorted->at(indexOfInsertedElem);
      insertPos : Integer = indexOfInsertedElem;

      while insertPos > 1
        begin
          if insertedElem < osToBeSorted->at(insertPos - 1) then
          begin
                update osToBeSorted->insertAt(insertPos, osToBeSorted->at(insertPos - 1));
                update insertPos = insertPos - 1;
          end
          endif
        end
     endwhile

     update osToBeSorted->insertAt(insertPos, insertedElem);
     update indexOfInsertedElem = indexOfInsertedElem +1;
    end
  endwhile
end
```

Listing 3.5: XOCL expression specifying the insertion sort algorithm

Before advancing to the next issue, the XOCL expressions addressed above are put together to specify the *Insertion Sort* algorithm as shown in Listing 3.5. We know that, OCL has a predefined *sort()* operation, which can be used to specify the sort intention on a given collection but leaving concrete sorting algorithm unknown at modeling time. The purpose of this example is to illustrate the action-language-enabled aspects of our XOCL extension. The ordered set, whose integer elements are to be sorted, is defined as local variable in an outermost block-expression. To keep things complete, a separate XOCL if-expression is nested into an XOCL while-expression, which in fact can be merged as the second condition into its containing while-expression. This example shows, with XOCL imperative expressions, complex and detailed behavioral modeling like this can be accomplished.

After extending OCL with new language constructs, the underlying abstract syntax element must be enhanced accordingly. Figure 3.19 shows the class hierarchy of XOCL abstract syntax elements. The most important OCL abstract elements are shown as classes with white background color, which are in fact our own Java implementations of the OCL abstract syntax described in

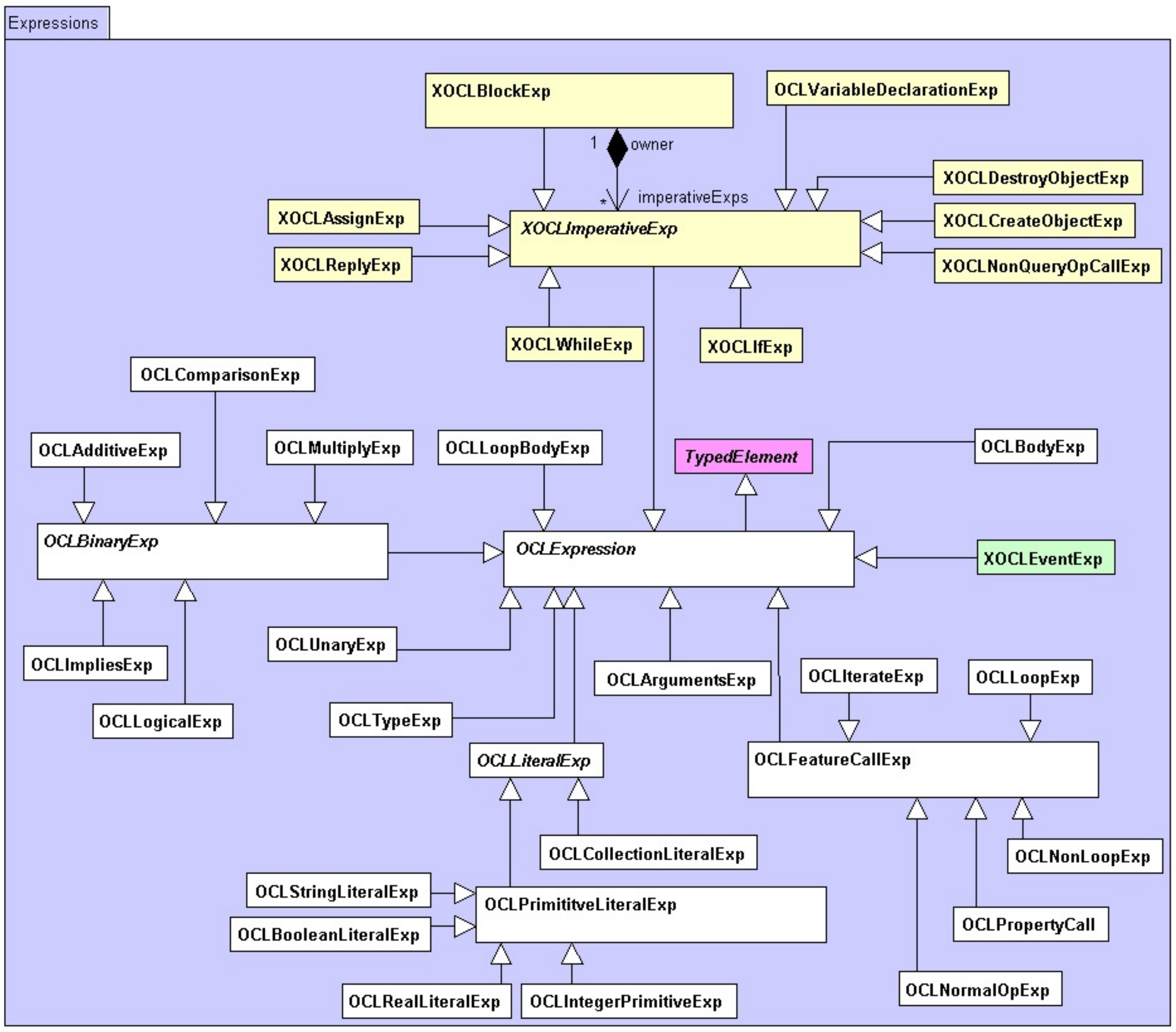


Fig. 3.19.: The class hierarchy of XOCL abstract syntax

[OMG10a] with slight modification in order to make the compiling process efficiently. All the XOCL abstract syntax elements in terms of imperative expressions are shown in light-yellow background color. The composition pattern is used again to depict the inclusion and possible nested relationship between an XOCL block expression and its contained imperative expressions. The single class in light-green background color represents the root of an abstract syntax tree of XOCL event expression. As defined in [OMG10a], all the OCL expressions are *typed elements* in UML. That means, evaluating an OCL expression results always in a type, which must be existent either in a predefined library, like our MOCCA DPM, or in the actual model. For the XOCL imperative expressions without meaningful type, like block-, while- and compound if-expression with block expression in its branches, the *OclVoid* is always assigned to the abstract syntax element representing them as the result type.

Besides the inheritance relationships shown in Figure 3.19, there are important associations between the XOCL abstract syntax elements as well as between them and UML meta model elements. Instead of listing all the class diagrams modeling these relationships, an abstract syntax tree (AST) generated by the XOCL parser, which is addressed in Section 4.2, is given in Figure 3.20 to illustrate the working principle of the XOCL abstract syntax implementation. The reason for adopting this strategy is that the original OCL abstract syntax has been documented well in [OMG10a] on the one hand, on the other hand, the small set of extended XOCL abstract elements are closely related to its concrete syntax in Listing 3.4, it is easy to find out the relationships between abstract syntax elements by understanding their concrete syntax.

Back to the given AST example, this AST represents the expression in Listing 2.2 with slight modification that explicit iterator variables, means, p and s are defined. Both subtrees rooting in *LP1* and *LP2* nodes with the AST type *OCLLoopExp* represent the OCL *select()* and *forAll()* operation, respectively. At the runtime both instances of *OCLLoopExp* hold the information to identify the individual OCL loop operation. The *OCLFeatureCallExp* owns generally two branches. The one on the left-hand side is the caller, whereas the other one is the callee. The callee usually uses the type information carried by the caller for type checking and code generation. Each object in yellow color represents a model element defined in the UML model. At the time of constructing an AST, each token, which may represent a model element, is type-checked against the symbol table, which is created based on the underlying UML model. If it is found, the corresponding model element will be associated with the AST node representing the current token. For example, the AST node *PC2*, which represents the token *programPartners*, has been linked with the association-end *programPartners* defined in the class diagram shown in Figure 2.9. It is to note that the same model element can be linked to different AST-nodes, that produces a circle in an AST. In fact, such a model element linked to AST-nodes should not be considered as a normal tree node, but an attribute of an AST-node.

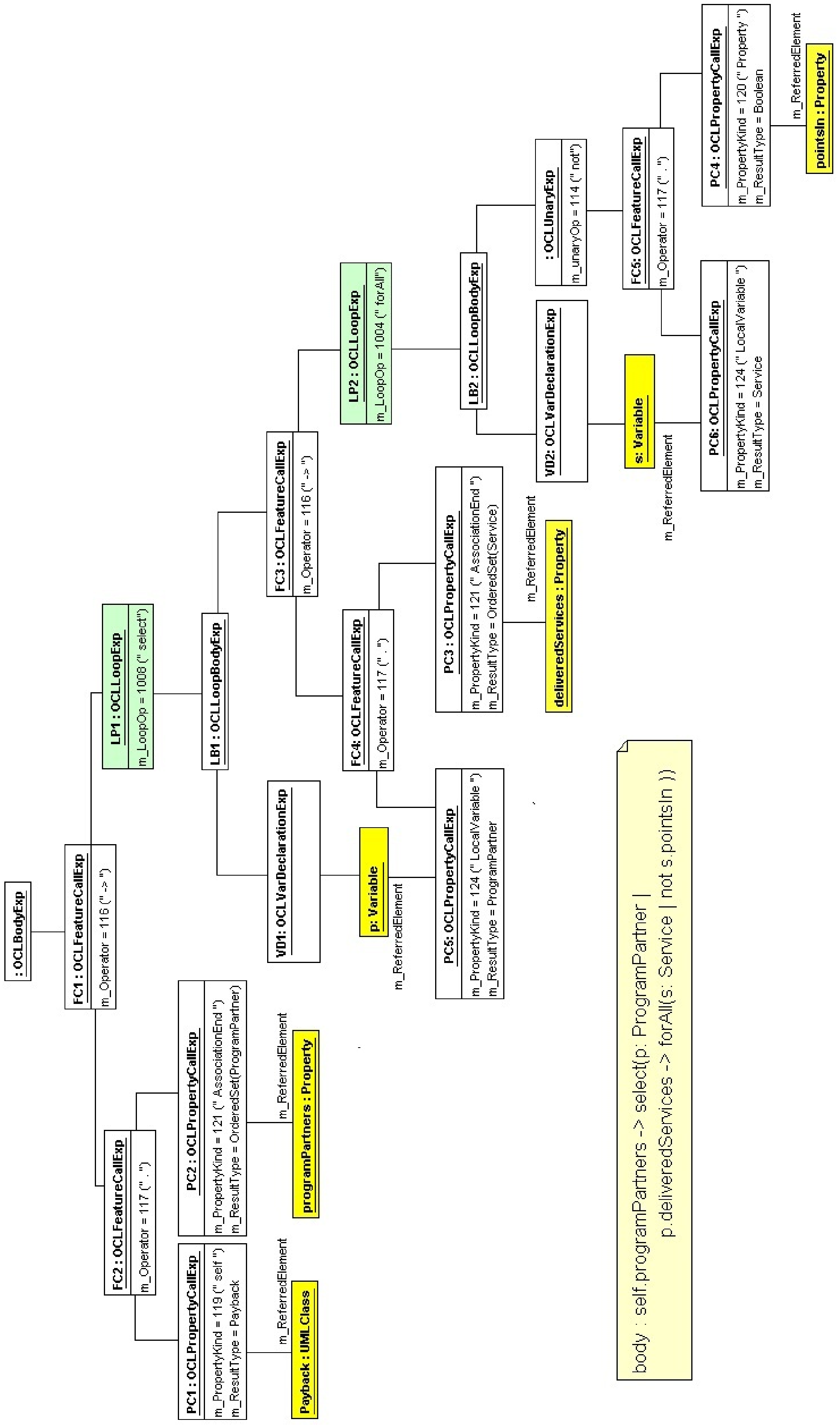


Fig. 3.20.: The abstract syntax tree of an XOCL expression

4. The MOCCA Model Compiler

4.1. The Working Principle of MOCCA Model Compiler

This chapter addresses the second issue raised in Section 1.2 - efficient model transformation and code generation for MDA. For years of research in this field, different algorithms and principles have been developed by Dr. Fröhlich and me. As a proof-of-concept of these methodologies, we developed our MOCCA model compiler in Java programming language. As a stand-alone application, MOCCA dose not rely on any hosting environment like those MDA tools residing in Eclipse ecosystem, but cooperates with other tools to obtain necessary configuration artifacts, to feed its output into other down stream tool-set for further processing, etc. Figure 4.1 gives an overview of the MOCCA development environment. Instead of using high-level context diagram, the UML activity diagram exploses more details of this environment, which are considered as helpful to understand the idea behind MOCCA. In Figure 4.1 significant developing tasks and their possible tool support are depicted in single action, whereas the required input and produced output are represented as object nodes.

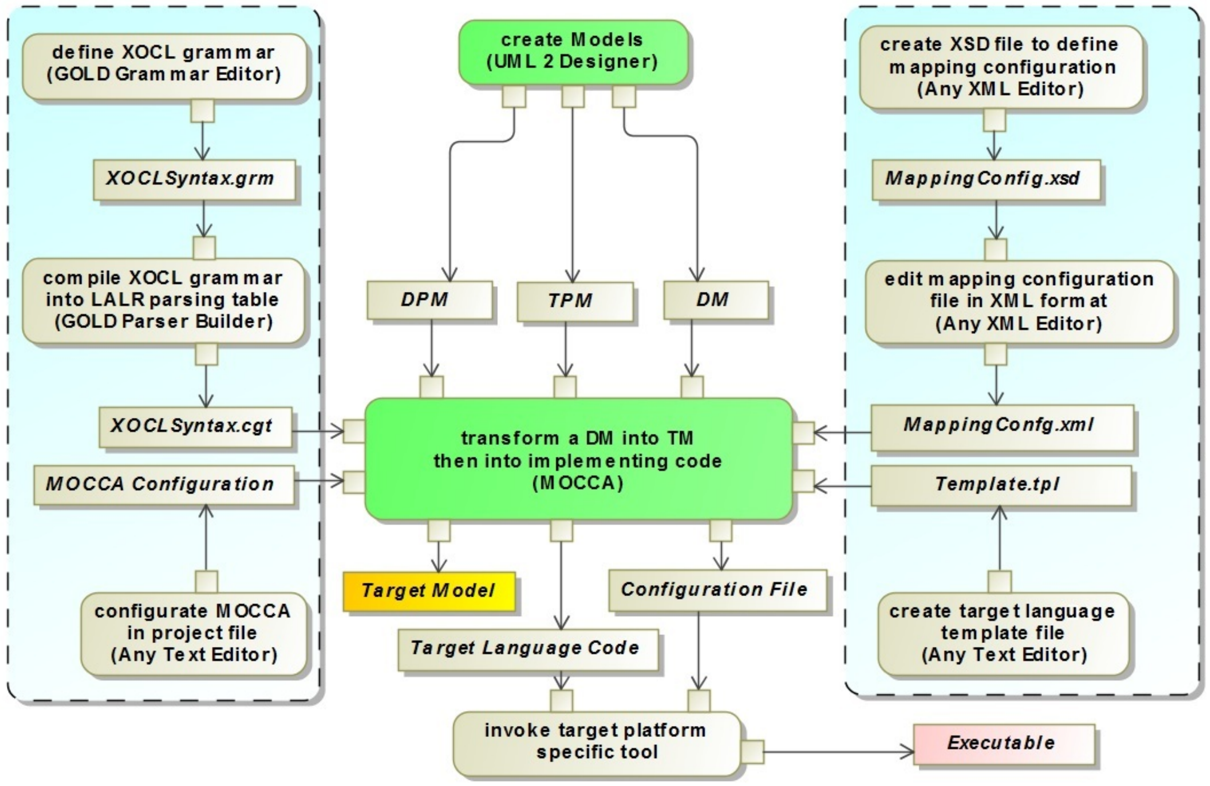

define XOCL grammar (GOLD Grammar Editor)
XOCLSyntax.grm
compile XOCL grammar into LALR parsing table (GOLD Parser Builder)
XOCLSyntax.cgt
MOCCA Configuration
configurate MOCCA in project file (Any Text Editor)
create Models (UML 2 Designer)
DPM
TPM
DM
transform a DM into TM then into implementing code (MOCCA)
Target Model
Target Language Code
Configuration File
invoke target platform specific tool
Executable
create XSD file to define mapping configuration (Any XML Editor)
MappingConfig.xsd
edit mapping configuration file in XML format (Any XML Editor)
MappingConfg.xml
Template.tpl
create target language template file (Any Text Editor)

Fig. 4.1.: MOCCA Development Environment

The UML 2 Designer is used to create all the three kinds of UML models for MOCCA as required inputs. Once the MOCCA design platform model (DPM) and target platform model (TPM) have been created, they can be reused for different projects, whereas the design models

(DM) are unique from project to project. The UML 2 Designer exports models either in binary format or in a dedicated relational database. MOCCA is also equipped with appropriate model readers to retrieve these models from their repository.

As already addressed, XOCL is the action language designed in this thesis to model behaviors in MOCCA DM. The XOCL grammar is edited using GOLD Grammar Editor [Coo13], which exports a *.grm* file. The GOLD Parser Builder then checks the XOCL grammar file and compiles the grammar rules into a LALR parsing table [Aho+08], which is used by XOCL parser component of MOCCA to check XOCL expressions involved in design model. Unless the XOCL will be extended or modified, the produced parsing table (saved in a *.cgt* file) is stable and reusable.

To initialize MOCCA, a global configuration file called *MOCCA Project File* is used to save important parametrization, whose items concern concrete compiler components to be used, project output folders, etc. This configuration file is also reusable for most cases.

On the right hand side of Figure 4.1 are artifacts in terms of configuring model transformation and code generation. An XML Schema Definition is created to regulate *MOCCA Mapping Configuration File*, which is an XML file and will be addressed later in this chapter. Similar to target platform model, for each concrete target platform, there is at least one mapping configuration file.

The primary objective of MOCCA is to generate applications implemented in OOPLs. Thus, MOCCA organizes the produced target language code into classes and interfaces. The *Target Language Template File* defines the class layout of a target OOPL. It is intuitive to understand that target language has its own template, but not target platform. For example, the Java Standard Edition (JSE), the Java Enterprise Edition (JEE) as well as Google Android API are considered as different target platforms naturally. However, developing applications atop all of them means producing code in Java programming language. Thus, one Java template file is shared by all of these target platforms.

Once all the required inputs are provided, MOCCA can transform a design model into its target model, which is highlighted in Figure 4.1 with yellow background color. Essentially speaking, the TM can be considered as some kind of *intermediate output* of MOCCA, because it only resides in memory and will be used for code generation directly. The TM is shown here to illustrate that MOCCA follows the MDA principle in terms of transforming a PIM into PSM then into code. After code generation, both target language code and possibly necessary configuration files are fed into target platform specific tools to generate an executable application as the last step.

With understanding of the MOCCA development environment, which builds the context for MOCCA and other involved tools, the following discussion will concentrate on MOCCA itself. Figure 4.2 illustrates the most important activities that are performed in a typical compilation flow. MOCCA conforms to the classical compiler architecture that consists of compiler front end and back end, is equipped with symbol table, etc [Aho+08]. The most actions and artifacts in Figure 4.2 are self descriptive and easy to understand. In the rest of this chapter, essential activities like model validation, model transformation and code generation will be addressed in their own sections in detail. For now, an overview is given as follows:

- *initialize Compiler* start-ups the compiler engine and other kernel components, such as compiler logger, compiler internal model repository, and compiler configurator, which parses the parameter-settings in project file to integrate appropriate components into the actual compilation.

- *parse PIM* reads the design model and design platform model into compiler internal model representation. Because a DM is always created based on the model elements from DPM, which is an imported module in DM. Thus, only the DM must be provided and a model reader separates the both models automatically.

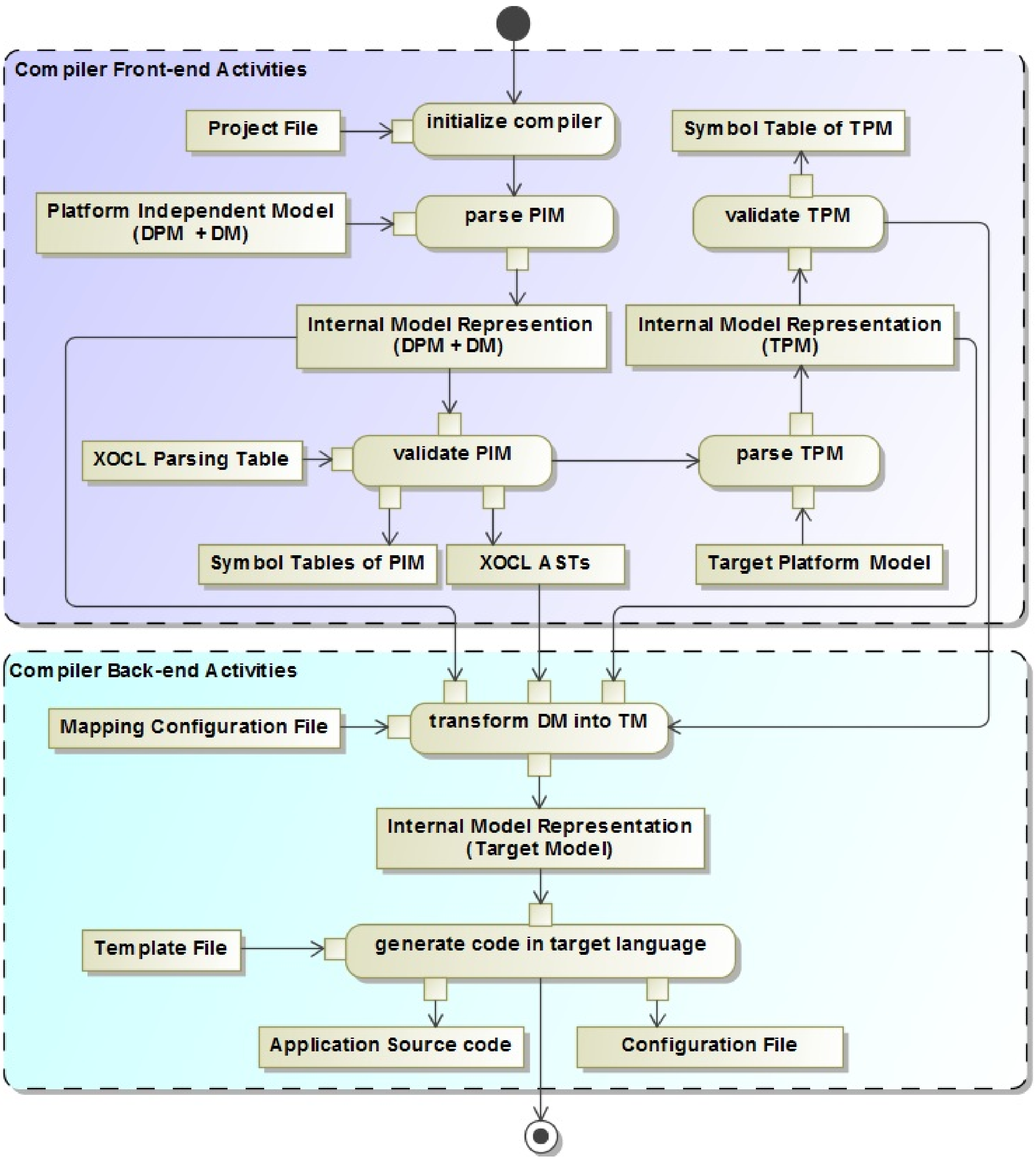

Compiler Front-end Activities
Project File
initialize compiler
Symbol Table of TPM
Platform Independent Model (DPM + DM)
parse PIM
validate TPM
Internal Model Represention (DPM + DM)
Internal Model Representation (TPM)
XOCL Parsing Table
validate PIM
parse TPM
Symbol Tables of PIM
XOCL ASTs
Target Platform Model
Compiler Back-end Activities
Mapping Configuration File
transform DM into TM
Internal Model Representation (Target Model)
Template File
generate code in target language
Application Source code
Configuration File

Fig. 4.2.: MOCCA Compilation Flow

- *validate PIM* sorts the model elements in class diagrams and records them into the symbol table in the form of *qualified name - model element* pair on the one hand. On the other hand, all the XOCL expressions specifying operation implementations in class diagram are parsed into the corresponding abstract syntax trees for sub-sequential processing.
- *parse TPM* and *validate TPM* repeats almost the same tasks as in the previous two steps.
- *transform DM into TM* is the most important and complicated operation in a complication flow. It executes the core algorithms implemented in the model mapper as well as consulting mapping rules defined in the mapping configuration file to map a platform independent

design model into its platform specific representation. Especially, all the behavior specifications in TM are already in target language, because the XOCL ASTs created in model validation have been traversed in a model mapping process for emitting target code.

- *generate code in target language* is the last step, in which the target model is traversed and target language code is generated and exported into external files together with necessary configurations.

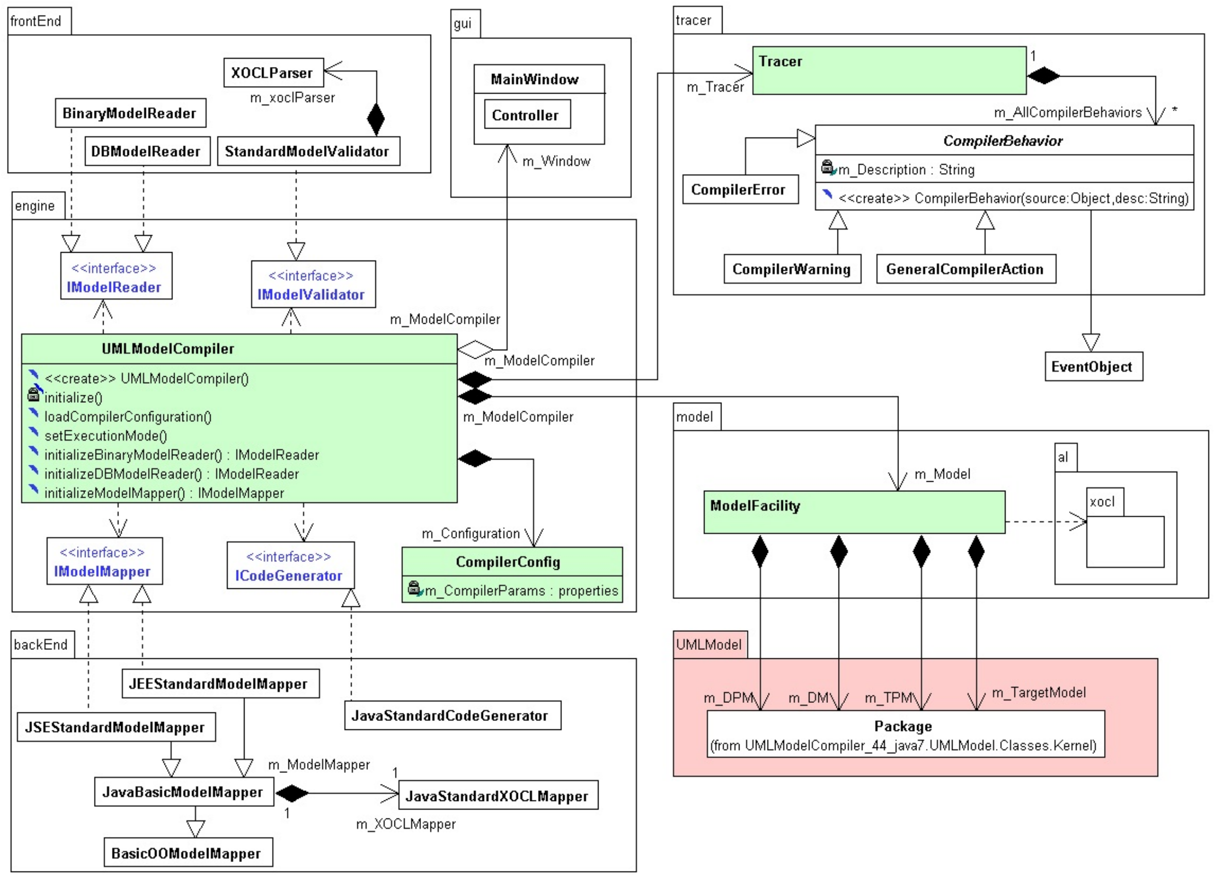
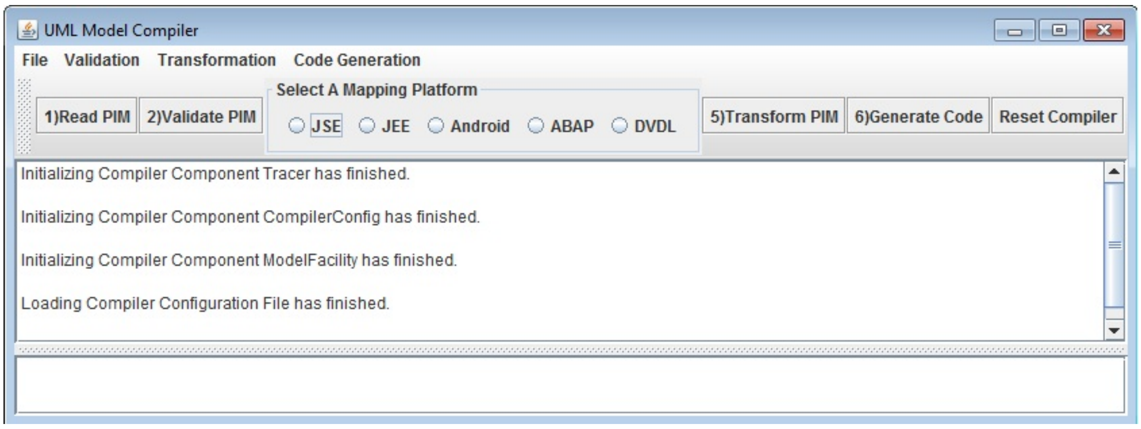


Fig. 4.3.: MOCCA Compiler Architecture

Before addressing development issues of each significant compiler component, an overview of relevant parts of the compiler architecture is illustrated in Figure 4.3. As mentioned earlier, the presented approach is backed by a flexible compiler architecture that enables the dynamic integration of user-specific components. Again, instead of using high-level component diagram, a class diagram with top-level packages for each relevant compiler component is given to explore some more details, which ought to be helpful to follow the overall compiler architecture.

All the implementation classes denoted with green background color will be bootstrapped immediately, which build the kernel framework of the compiler. After start-up, MOCCA is ready for integrating other functional components to accomplish the actual compiling process. The *ModelFacility* class is the internal model repository, which stores all the models imported and generated by accessing both the *UML Metamodel* sub-component and the *XOCL Abstract Syntax* sub-component. Moreover, *ModelFacility* maintains symbol table for model elements and provides powerful query mechanisms for efficient model transformation. The *Tracer* component logs all the sensitive compiler-behaviors in the form of object hierarchy, instead of plain text. Besides the three typical compiler behaviors shown in Figure 4.3, namely, *CompilerError*, *CompilerWarning* and *GeneralCompilerAction*, there are also component-specific behaviors, which are not shown here due to restriction of place.



```
Initializing Compiler Component Tracer has finished.

Initializing Compiler Component CompilerConfig has finished.

Initializing Compiler Component ModelFacility has finished.

Loading Compiler Configuration File has finished.
```

Fig. 4.4.: The initial GUI of MOCCA after bootstrap

MOCCA was designed as a console application in [Frö07]. If MOCCA is started, it runs from begin to the end. This one-way working principle has a remarkable shortcoming - even error occurred only in one compiling phase, the entire compilation must be done again. To cope with this dilemma, the new generation of MOCCA is equipped with a light weight GUI interface, which provides the intuitive interaction and hence, eases the compilation process. Figure 4.4 illustrates the MOCCA GUI, which displays the initial status of MOCCA immediately after bootstrap. The main working area of this GUI is divided into the upper region for displaying normal compiler behaviors as well as compiler warnings, and the bottom region reserved only for compiler errors. The order and semantics of the toolbar buttons reflect the compilation flow depicted in Figure 4.2. The operation according to the exclusive radio buttons for selecting target platform merges the compiler actions for reading and validating TPM.

After the initialization of the compiler framework, the main functional components can be connected to the compiler engine based on the current project configuration. The compiler front-end takes over the responsibilities of importing UML models from external repositories into native model representation, establishing symbol tables as well as parsing XOCL expressions into their AST-representations. The back-end components realize the model transformation and code generation. All the functional components in charge of a concrete mission couple to the underlying engine via interface implementation loosely. This architecture leads to a flexible system with wide spectrum of interchangeable functional components. Figure 4.3 shows a particular configuration of MOCCA, in which *BinaryModelReader* for parsing models in binary format and *DBModelReader* for parsing models saved in a relational database are provided, *StandardModelValidator* and its *XOCLParser* are used, and in back end, both the model mappers for JSE and JEE are given together with a common Java code generator. Model transformation and code generation for other platforms can be supported by extending the back end with additional model mappers and code generators.

4.2. Model Parsing and Validation

MOCCA is designed primarily to work with our own CASE tool UML 2 Designer, which can persist models either in binary format via *Java Object Serialization* [HC08] or in a dedicated relational database running on both MySQL and Oracle. The model reader components equipped by MOCCA conform to those that are used by UML 2 Designer. Detailed technical documentations about model import and export of UML 2 Designer are provided as according project reports, and moreover, due to restriction of space, they cannot be explained again. To understand the MDA approach presented in this work, the only important thing is to be aware that model parsing returns the object representing the top-level package of a UML-modeling project along with all its subordinate model elements. As mentioned before, the design model contains application model itself as well as design platform model as imported module. The model parsing for PIM separates the both models and assign them to *m_ DPM* and *m_ DM* in *ModelFacility* as shown in Figure 4.3. It is feasible, because according to MOCCA modeling framework, the UML package dealing with design model and design platform model have to be labeled by stereotypes *«DesignModel»* and *«DesignPlatformModel»* respectively.

As the next step, a model validator checks all the elements in the parsed model, which may participate in the following model transformation, to record them into appropriate symbol table for efficient query. MOCCA is lunched with only one *StandardModelValidator* together with its default *XOCLParser*, as illustrated in Figure 4.3. However, arbitrary sophisticated validation logic can be involved by either sub-typing the standard model validator or by decorating it via *object composition*. These topics belong to extending MOCCA that will not be addressed in this work in detail. In this section the discussion limits to the preliminary checking in standard model validator.

Algorithm 4.1 Validate design model elements and record them into symbol table

```
Require: Package representing the parsed design model: $dm$
Ensure: Symbol table for DM elements and ASTs of operation-behaviors
 1: for all $PackageableElement\ pe \in dm$ do
 2:     if $pe$ is $Package$ then
 3:         validate $pe$ as package
 4:     else if $pe$ is $UMLClass$ then
 5:         validate $pe$ as class
 6:     else if $pe$ is $DataType$ then
 7:         validate $pe$ as data type
 8:     else
 9:         validate $pe$ as interface
10:     end if
11: end for
12: for all $UMLClass\ cl \in dm$ do
13:     for all $Association\ ass$ coming from or ending at $cl$ do
14:         validate $ass$ with $cl$ as its context
15:     end for
16: end for
17: for all $Operation\ op \in dm$ do
18:     if $op$ has $Behavior$ then
19:         validate $op$'s first behavior only
20:     end if
21: end for
```

Algorithm 4.1 exposes the high-level routines validating the parsed DM model elements. Validation of DPM as well as TPM model elements just follows the same strategy. This algorithm is considered as high-level due to its *dispatcher*-characteristics. In the first loop, all the UML *PackageableElements* [OMG10b], whose direct container package is the design model itself, are processed. Based on their concrete type, they are dispatched to the responsible validation routines. It is to note that the routine to validate a package in line 3 can be called recursively for its sub-packages. Because *UMLClass*, *DataType* and *Interface* are *NameSpaces* [OMG10b], which can contain other elements like *Operation* and *Property*, etc. Hence, the validation routines for them also start validations for such contained elements, if necessary. After execution of the first loop in Algorithm 4.1, the *Hashtable*-based symbol tables listed in lines 5 to 13 in Listing 4.1 have been filled with data in the form of *full name* $\rightarrow$ *model element* pair.

```
 1 package compiler.model;
 2
 3 public class ModelFacility
 4 {
 5   private Hashtable<String, Package> m_SymbolTablePkg;
 6   private Hashtable<String, UMLClass> m_SymbolTablecls;
 7   private Hashtable<String, DataType> m_SymbolTableDT;
 8   private Hashtable<String, Property> m_SymbolTablePty;
 9   private Hashtable<String, Operation> m_SymbolTableOp;
10   private Hashtable<String, Interface> m_SymbolTableIF;
11
12   private Hashtable<String, Profile>m_SymbolTablePr;
13   private Hashtable<String, Stereotype>m_SymbolTableSt;
14
15   public void registerModelElement(String fullName, Element e);
16
17   public UMLClass   findClassByName(String fullName);
18   public DataType   findDataTypeByName(String fullName);
19   public Interface  findInterfaceByName(String fullName);
20   public Operation  findOperationByName(String fullName);
21   public Property   find PropertyByName(String fullName);
22   public Stereotype findStereotypeByName(String fullName);
23
24   public int getDistanceBetweenClassifiers(Classifier subCls, Classifier baseCls);
25 }
```

Listing 4.1: Symbol table and the according operations defined in model repository component

Because profile is a sub-type of package, the symbol table for *Profile* is filled by routine validating packages, whereas the routine validating classes fills data in the *Stereotype* symbol table with the same reason.

The second loop in lines 12 to 16 in Algorithm 4.1 resolves all the associations with the current context class *cl* as one of its association-ends. For each association-end that is *navigable* from the current context class, an additional property is recorded into symbol table *m_SymbolTablePty*.

In the last loop of Algorithm 4.1, *XOCLParser* is called to parse the XOCL expressions in the opaque behavior, which specifies the implementation of an operation in the design model. If an operation has more than one behavior, only the first one will be taken into account, the others will be ignored. Parsing behaviors as the last step in model validation is reasonable. Because one of the most important jobs to do in parsing XOCL is the type checking against the underlying UML models. The type checking can be done efficiently by using the query-operations defined in lines 17 to 22 in Listing 4.1 after that the classified symbol tables have been established completely.

The most complicated work in model validation is to parse the XOCL expressions to construct their abstract syntax trees containing all the necessary semantic information for the next step of model transformation. The *XOCLParser* component, whose general structure is given in Listing 4.2, takes over this responsibility. As mentioned in Section 3.4 and further addressed in Section 4.1, to simplify the development of an XOCL parser, the GOLD parsing system is used as the compiler generator, which supports the LALR(1) grammar [Aho+08] and provides the tools to edit the grammar rules and transform grammar rules into LALR parsing table. Up to this step, all the work done is completely implementation language independent, but only concerns mathematics. Thanks to Matthew Hawkins, a Java Engine [Haw13] for GOLD has been developed, which contains a language parser along with concrete syntax tree API in Java.

```
 1 package compiler.frontEnd.modelValidator;
 2
 3 public class XOCLParser implements GPMessageConstants
 4 {
 5   private GOLDParser m_Parser;
 6   private Reduction m_CSTree;
 7   private XOCLAST m_ASTree
 8   private ModelFacility m_Model;
 9
10   public void parseBehaviorForOperation(String xoclCode, Operation context);
11   public void constructConcreteSyntaxTree();
12   public void constructAbstractSyntaxTree();
13
14   private void initialize();
15   private OCLExpression CSTreeWalker(Reduction cstNode, IType context);
16   private XOCLCollection deriveCollectionType(boolean isUnique, boolean isOrdered);
17   private int typeChecker(FeatureSignature symbol, IType callerType, OCLFeatureCallExp
        astNode);
18 }
```

Listing 4.2: The class framework of the XOCLParser

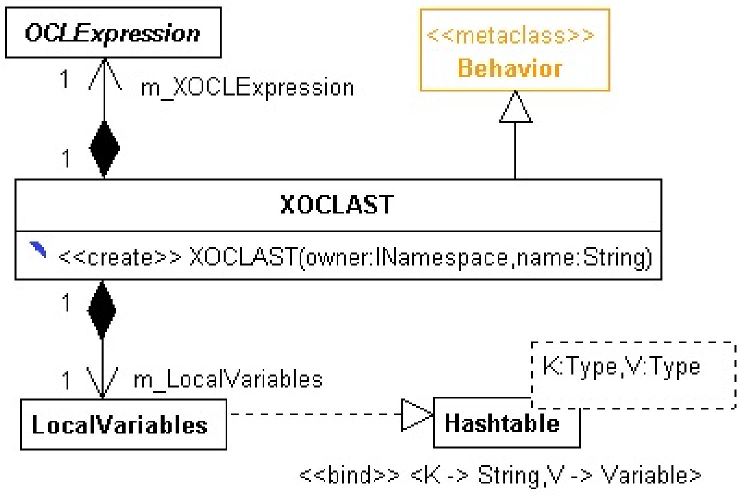


Fig. 4.5.: Wrapper class *XOCLAST* representing the root of an XOCL abstract syntax tree

With the favor of the Java Engine, the *XOCLParser* references to a *GOLDParser*-instance shown in line 5 of Listing 4.2, whose *parse()* operation is called in the *constructConcreteSyntax-Tree()* operation (line 11 of Listing 4.2) and parses an XOCL expression given by the *parseBe-haviorForOperation()* operation (line 10 of Listing 4.2) by consulting the LALR parsing table saved in *XOCLSyntax.cgt* file and installed by the *initialize()* operation (line 14 of Listing 4.2) into its concrete syntax tree saved in the parser property *m_ CSTree* in line 6 of Listing 4.2. The *Reduction* API class, which is the data type of *m_ CSTree*, encapsulates the *reduce* operation [Aho+08] in LR parsing with the grammar rule used as well as all the grammar symbols involved in the current reducing. Hence, a reduction to the *start symbol* indicates the successful parsing, meanwhile maintains the root of the constructed concrete syntax tree.

After the concrete syntax tree is constructed, there is no more syntactical error in the parsed XOCL expression. As the next step, the *constructAbstractSyntaxTree()* operation in line 12 of Listing 4.2 tries to build the abstract syntax tree by top-down traversing each CST-node representing a none-terminal grammar symbol. If no further error appears, the above operation constructs an *XOCLAST*-object and saves it in the parser property *m_ASTree*. As Figure 4.5 illustrates, the *XOCLAST* is a wrapper class, which references to a top-level XOCL abstract syntax element that either represents an OCL-body expression or an XOCL-block expression addressed in Section 3.4. Instead of preserving local variables globally in model facility, this class maintains an additional symbol table for the resolved local variables, which are only valid in the scope of one XOCL-expression. Just like the *OpaqueBehavior*, the *XOCLAST* is also derived from the metalclass *Behavior*. Thus, after constructing AST, it replaces the original opaque behavior of an operation for further model transformation.

Algorithm 4.2 Check the data type of a node in CST, which may represent an identifier

Require: Identifier with additional information: *symbol*. Possible context in which *symbol* may be defined: *callerType*. The AST node representing the actual identifier: *astNode*

Ensure: Model element corresponding to *symbol* is found and connected to *astNode*.

```
 1: if symbol is Operation then
 2:     check if symbol is inherited from OclAny
 3:     if callerType is one of the OCL primitive data types then
 4:         check if symbol is OCL predefined operation
 5:     else
 6:         $opSig \leftarrow callerType.fullName + symbol.signature$
 7:         try to find opSig in the symbol table of operations
 8:         if opSig not found then
 9:             call findMSO routine to find a most specific operation matching symbol
10:         end if
11:         if symbol still not matched as an operation then
12:             for all direct super-type superCallerType of callerType do
13:                 call the same type-checking routine with superCallerType as parameter
14:             end for
15:         end if
16:     end if
17: else
18:     try to resolve symbol as LocalVariable
19:     if not found then
20:         try to resolve symbol as Parameter defined in the current context operation
21:     end if
22:     if not found then
23:         try to resolve symbol as Property defined in callerType
24:     end if
25:     if not found then
26:         for all direct super-type superCallerType of callerType do
27:             call the same type-checking routine with superCallerType as parameter
28:         end for
29:     end if
30: end if
```

During the top-down traversal of CST, information must be passed up and down between CST-nodes. This information exchange is usually realized by using *synthesized* as well as *inherited* attributes [Aho+08]. Different with those *recursive descent* parsing system in that both inherited and synthesized attributes can be treated simply as input parameter and return value of an operation denoting a grammar production, a helper operation *CSTreeWalker()* is provided in our *XOCLParser*, whose return value denotes the common synthesized attribute for a constructed

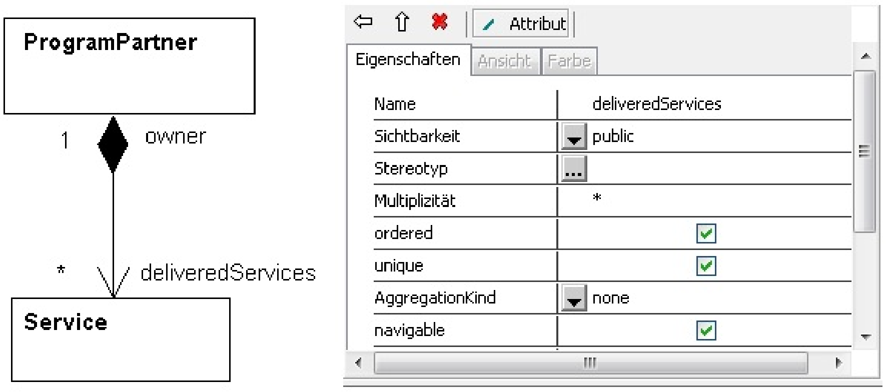
XOCL AST node and second input parameter is the generally required inherited attribute, because different none-terminal symbols can have extra inherited attributes. For such situations, additional stack-based parser properties will be used to store information for the next-level tree-traversal.

The most important issue during construction of AST is the type checking together with filling AST node with model element defined in the structural model in class diagram. The type checking routine is implemented in the *typeChecker()* operation in line 17 in Listing 4.2. The first parameter with the type *FeatureSignature* denotes a wrapper class that encapsulates the name of an *identifier* with additional information, like whether it concerns an operation or other typed element. In the case of an operation, all the arguments used by calling this operation are involved in this wrapper. The second parameter is the *initial* type context, in which the identifier may be defined. The last parameter is the AST node, to which the resolved model element will be attached. Algorithm 4.2 explores the most necessary details of the type checking routine implemented in MOCCA.

Algorithm 4.3 Find the most specific operation

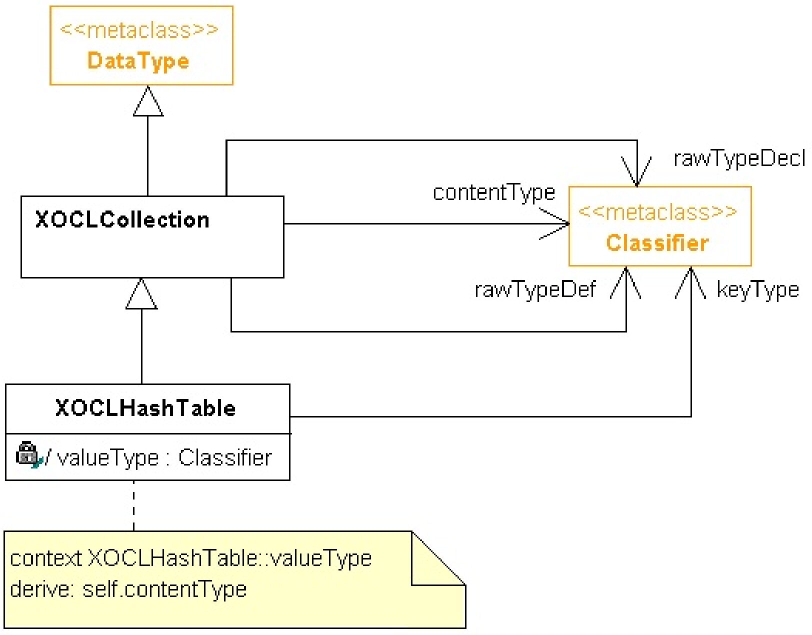
```
Require: Identifier representing an operation: $symbol$. Context in which $symbol$ may be de-
    fined: $callerType$.
Ensure: Operation $msop$ defined in $callerType$ and most specifically matching $symbol$
 1: retrieve $argList$, $argNumber$, $opName$ from $symbol$
 2: $msop \leftarrow \emptyset$
 3: $opCandidate \leftarrow$ operations defined in $callerType$, named $opName$ and having $argNumber$
    parameters
 4: for all $op \in opCandidate$ do
 5:     $paramList \leftarrow$ list of parameters of $op$
 6:     $searchFlag \leftarrow true$
 7:     for $i \leftarrow 1, argNumber$ do
 8:         $argDT \leftarrow$ data type of argument at position $i$ in $argList$
 9:         $paramDT \leftarrow$ data type of parameter at position $i$ in $paramList$
10:         $searchFlag \leftarrow searchFlag \wedge (distance(argDT, paramDT) >= 0)$
11:     end for
12:     if $searchFlag$ then
13:         $msop \leftarrow op$
14:     end if
15: end for
```

Because at the very beginning it is aware if an identifier concerns an operation or other typed elements, the type checker is able to make a difference between them that is reflected in Algorithm 4.2 with outermost *if* and *else* branches. To search an identifier, the underlying type hierarchy must be also taken into account. Both inner *for*-loops in lines 12 to 14 as well as in lines 26 to 28 in Algorithm 4.2 try to search an identifier in all the super types of the current type context in a recursive manner. For type-checking identifier as operation, if no absolutely matched operation can be found, a further effort will be made to find a *most specific* operation. This logic is illustrated in line 9 of Algorithm 4.2 by calling an extra routine called *findMSO*, whose logic of computation is shown in Algorithm 4.3 in detail. For the list of the argument types, an operation with the best corresponding list of parameter types is searched. In the process of finding the most specific operation, the *distance* of the actual argument type to the parameter type of a candidate operation is computed as shown in line 10 of Algorithm 4.3. The distance is a relation on types. If the both types are identical, the distance is 0, whereas for two types, they do not have any relationship in terms of inheritance hierarchy, the distance is defined as -1. Otherwise, the distance refers to the minimum number of gereralization relationships between them. The *distance*-function is implemented in the *getDistanceBetweenClassifiers()* operation as defined in line 24 of Listing 4.1 on page 81.



Name	deliveredServices
Sichtbarkeit	public
Stereotyp	...
Multiplizität	*
ordered	✓
unique	✓
AggregationKind	none
navigable	✓

Fig. 4.6.: Interpreting association-end as XOCL collection



```
context XOCLHashTable::valueType
derive: self.contentType
```

Fig. 4.7.: Wrapper classes handling XOCL collection types

In the process of type checking, the XOCL collection types can be resolved by interpreting the values set to the meta properties of a typed element, such as attributes, association-ends (both of them are referred as *property*, but the difference is made here to make things clearer) and operation parameters. The exact interpretation rules are given in Table 2.1 on page 34 and implemented in the *deriveCollectionType()* operation as defined in line 16 in Listing 4.2. Figure 4.6 shows a concrete setting in UML 2 Designer, which makes the association-end *deliveredServices* be interpreted as *OrderedSet* containing only objects with data type *Service* by the type-checker component. This semantics can be formulated as *lv: OrderedSet(Service)* for defining a local variable in the XOCL-expression. As addressed in Section 3.3.1, interpreting and using XOCL collection type as above concretize a template-based collection raw type defined in MOCCA DPM with its content type. This concretization can be represented in UML using *template-binding* [RQZ07], which involves relatively complicated meta model structure.

Because all the essential information for the type checking and the following model transformation are the collection raw type and its content type. Moreover, the internal representation of a concretized XOCL collection type is only checked in model validation and processed within the model transformation and never shown in UML graphical model. Hence, two additional wrapper classes are introduced to maintain all the necessary information at the meta level as depicted in Figure 4.7. The *XOCLCollection* is a *DataType* and associates three times with *Classifier* to record the raw type declaring the current collection, the raw type defining the current collection as well as its content type respectively. In PIM, only the raw type declaration is used, whereas in PSM both the raw type declaration and definition can be used, such that the former specifies the type for properties in classes and the latter is used to initialize them. The *XOCLHashTable* is an *XOCLCollection* with an extra association to *Classifier* to denote the *key type*. As the OCL-derive expression specifies, the *value type* is just a redefinition of *content type*.

```java
public class ProgramPartner
{
  private List<Service> deliveredServices;

  public ProgramPartner(){
       this.deliveredServices = new ArrayList<Service>();
  }
}
```

Listing 4.3: Example of mapping XOCL OrderedSet into Java

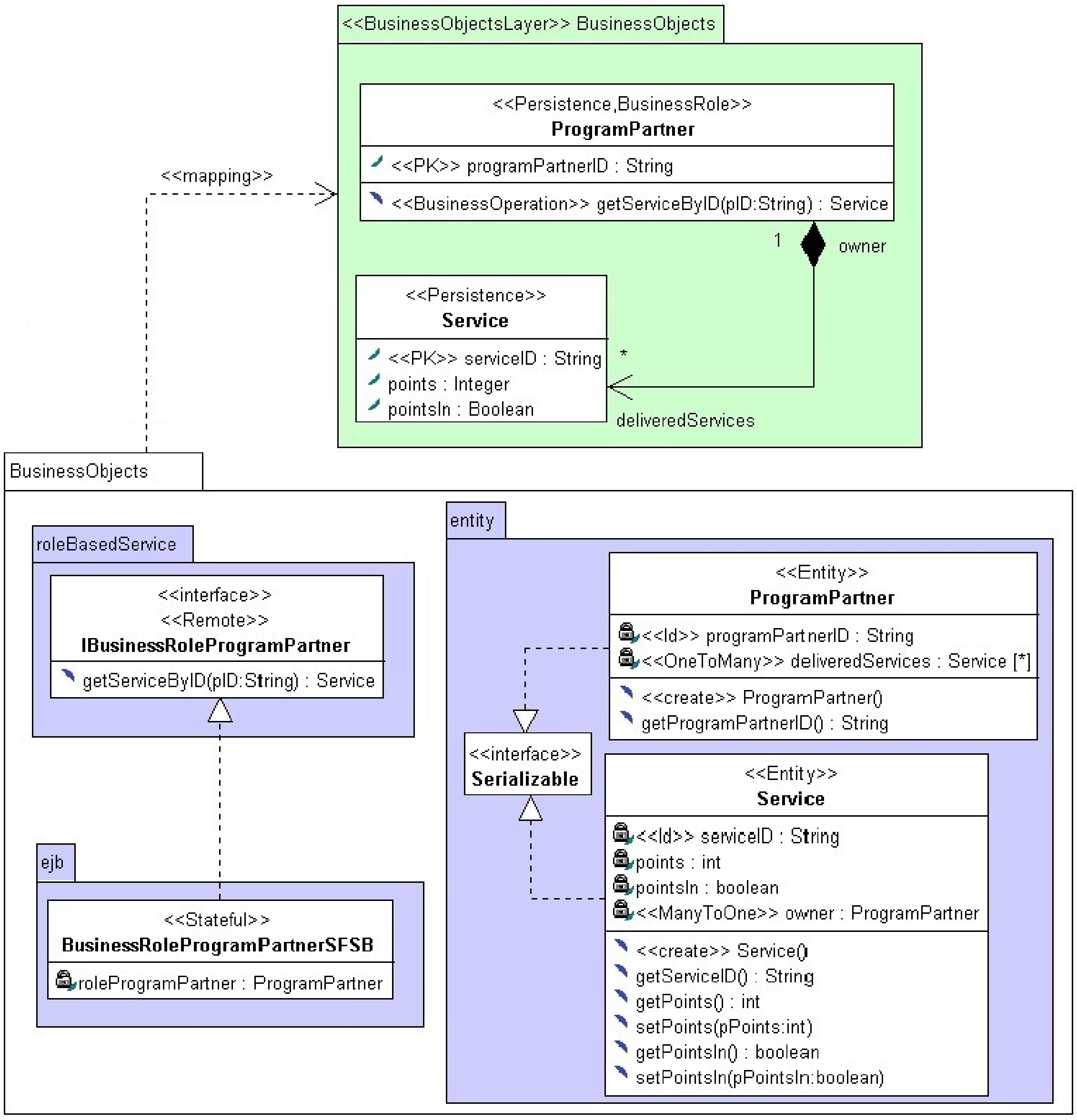
Listing 4.3 illustrates a possibly generated Java class, which highlights the association-end *deliveredServices* in the design model shown in Figure 4.6 after code generation. In type checking, an *XOCLCollection* instance referencing the *OrderedSet* read in from MOCCA DPM and *Service* modeled in the DM is created. In the following model transformation, the *Java Model Mapper* finds out that the *OrderedSet* ought to be mapped by *List<T>*-interface to declare an attribute but mapped by *ArrayList<T>* to initialize the according attributes by consulting the mapping rule table, and finally, creates another *XOCLCollection* instance to record all those information in the resulted target model for code generation.

4.3. Model Transformation

4.3.1. General Consideration of Model Transformation

As given in [OMG03] and introduced in Section 2.3.2, the model transformation is the pillar of the entire MDA methodology, but defined by OMG in a very abstract manner in order to give tool vendors more freedom to develop innovative technology realizing model transformation. This strategy results in wide range of researches and diverse methods and tools supporting model transformation, which have been summarized in Section 2.4 briefly. Despite the diversity of model transformation principles developed, the primary objective of model transformation is to *map* the platform independent *design model*, which is created by using the modeling facility provided by design platform model, into a *target model*, which is backed by the resources as well as services abstracted in the form of target platform model.

UML is a graphical modeling language and the models created in UML can be considered as *graph* according to mathematics. Hence, the *mapping* operation, which is at the heart of model transformation, can be defined as a *graph morphism*, $mapping : G_1 \rightarrow G_2$ [EPE06], in which G_1 is the design model whereas G_2 is the according target model. There are outstanding researches studying this mapping from the perspective of graph theory as described in [Sch94] [And+99] and [EPT04]. In this thesis, the mapping is discussed only from the view of software engineering and it is to avoid involving complicated and comprehensive mathematical derivations.



<<BusinessObjectsLayer>> BusinessObjects
<<Persistence,BusinessRole>>
ProgramPartner
<<PK>> programPartnerID : String
<<BusinessOperation>> getServiceByID(pID:String) : Service
1
owner
<<Persistence>>
Service
<<PK>> serviceID : String
points : Integer
pointsIn : Boolean
*
deliveredServices
<<mapping>>
BusinessObjects
roleBasedService
<<interface>>
<<Remote>>
IBusinessRoleProgramPartner
getServiceByID(pID:String) : Service
ejb
<<Stateful>>
BusinessRoleProgramPartnerSFSB
roleProgramPartner : ProgramPartner
entity
<<Entity>>
ProgramPartner
<<Id>> programPartnerID : String
<<OneToMany>> deliveredServices : Service [*]
<<create>> ProgramPartner()
getProgramPartnerID() : String
<<interface>>
Serializable
<<Entity>>
Service
<<Id>> serviceID : String
points : int
pointsIn : boolean
<<ManyToOne>> owner : ProgramPartner
<<create>> Service()
getServiceID() : String
getPoints() : int
setPoints(pPoints:int)
getPointsIn() : boolean
setPointsIn(pPointsIn:boolean)

Fig. 4.8.: A mapping transforms a piece of design model into its JEE target model

To expose the working principle of mapping, a concrete example is shown in Figure 4.8. In this simple demo example, the G_1 graph is a piece of design model, which models a *ProgramPartner* and its delivered *Service*s in a customer payback system. The G_2 graph is the corresponding target model based on JEE platform. Finally, the mapping itself is only represented as a connection between them on a very high-level of abstraction. It is to note that the *mapping* operation in Figure 4.8 has the direction from G_2 to G_1. The reason is that the mapping is modeled here as a stereotyped dependency, which has similar semantics as the predefined UML *«refine»* [OMG10b] stereotype. In such a dependency, the G_1 graph is the supplier, whereas the G_2 graph is the client refining its supplier with more platform details. Hence, the arrow of the mapping is on the side of the G_1 graph, namely, the source graph.

It is clear that the design model is much more compact than its target model, simply by counting the total number of nodes and edges involved in the both graphs. The most complexities resulted in target model are caused by involving technical details of implementation. Such details are generated in the *mapping* by interpreting markers that can be used to decorate nodes and edges. Markers are stereotypes and their tagged values in UML. Mappings targeted to different platforms are able to interpret those markers appropriately based on the technical as well as implementation requirements on one concrete target platform. For example, a mapping targeted to JEE platform maps each design model class stereotyped by *«Persistence»* into a Java class annotated by *Entity* annotation, which is represented in Java TPM as *«Entity»* stereotype, realizing the *Serializable* interface, containing a default constructor with empty parameter list and providing *getters* for all properties and *setters* for all non-read-only properties with public access control in design model.

It is worth noting that the nodes in both graphs are not simple as defined in classical graph theory, but structured with sub-nodes. Typical sub-nodes are properties and operations and moreover, the latter itself can have sub-nodes - operation parameters, etc. Properties and parameters are *typed elements* with assigned data type, according to UML terminology. As addressed in Section 3.3, the MOCCA DPM provides diverse pre-defined data types for creating design model. To map them into the target platform counterparts, the according mapping rules should be given and accessible to the mapping. As illustrated in Figure 4.8, for example, to map the DPM primitive type *Integer* to Java primitive type *int*, a mapping rule should be defined in the mapping configuration file, which will be addressed later in this chapter.

UML model is defined by its metamodel. Hence, the both models in Figure 4.8 are maintained in the form of in-memory instance graph with each node typed by an appropriate UML metaclass within the both modeling and MDA tool. Therefore, from the perspective of a practical implementation of *mapping*, it can be considered as a graph morphism, which transforms an internal metamodel instance graph of design model into the according metamodel instance graph of target model.

The example in Figure 4.8 concerns only one aspect of mapping - the *structural mapping*. The other one is the *behavioral mapping*, which traverses each XOCL abstract syntax tree rooted in an *XOCLAST*-object maintained by operation, to emit implementation code in target language and saved again in an opaque behavior, which replaces the previous *XOCLAST* object as the final behavior in target model.

Up to this point, the significant features and characteristics of *mapping* can be summarized as follows:

1. For one input design model, different mappings are to be developed, each of which is conceived to a target platform.

2. A mapping is able to transform design models based on predictable object-oriented principles, such that for each class in design model, a class in target model is to be generated at least, etc.

3. A mapping is able to interpret DPM stereotypes applied to a design model element appropriately, if such stereotypes are meaningful for that target platform.

4. Connections between data types in DPM and their counterparts in TPM are defined in an external mapping configuration file. A mapping is able to resolve the typed elements in design model by consulting mapping rule table, which records mapping rules for platform-data-types.

5. A mapping is able to traverse the *XOCLAST*-objects for emitting target language codes either by itself or delegating this traversal to other component in charge.

4.3.2. Target Platform Issues

Before the design and development issues of a concrete model mapper are addressed, several crucial aspects common to all the potential target platforms must be considered well in terms of finding reasonable and efficient mapping strategy to develop a model mapper for that target platform. As introduced in the previous section, especially, depicted in Figure 4.8, the typical output of a model mapper is a target model, which is completely based on the *building elements* provided by the underlying target platform, conforms to the *programming paradigm* of the underlying target platform as well as fine-tuned due to the *technical constraints* coming with the target platform.

Theoretically speaking, a correct design model can be mapped onto any target platform, which represents a general purpose programming language like C or Java, with the former a universal procedural programming language, whereas the latter a universal object-oriented programming language. It is obvious that the more convergent the design platform and a possible target platform are, the more straightforward a model mapper for that target platform can be developed. In the ideal situation, for each model element provided by the design platform, a direct counterpart can be found on an ideal target platform, moreover, both platforms share the same level of abstraction, the model mapper for that target platform needs only to execute *creating* counterparts for the design model elements in the target model as well as *substituting* design platform elements with suitable target platform elements. It is clear that there are always divergences between the design platform with high-level abstraction and the most practical target platforms. Thus, the model mapper is to be developed to smooth the differences between them. However, the *grade* of the divergence between the both essential platforms must be regulated in the scope of this work to define a reasonable context, in which the model mapper and the overall model transformation can be done. The following considerations are made to the target platforms supported by MOCCA.

- A target platform must base on an object-oriented programming language.

This requirement serves as the baseline for the model transformation addressed in this work. Exactly speaking, the model mappers involved in MOCCA won't deal with shifting programming paradigm. Because from the very beginning, the object-orientation is chosen as the central modeling and developing paradigm. Hence, the programming language like C, mentioned in the previous paragraph, is excluded as a possible target platform for the current version of MOCCA[1].

- Within the same implementation language platform (the classification in Section 3.2), the most suitable API or Framework sub-platform must be used to map the domain specific modeling constructs involved in a design model.

The application modeled in its design model is usually domain specific, which leverages the high-level, domain specific modeling constructs provided by the design platform. Before a potential model mapping starts, the target platform reflecting the chosen implementation language is already known. However, for a universal OOPL like Java, there are diverse APIs as well as frameworks, which may better suit the design model and ease the mapping process by providing language constructs with similar abstraction level and domain specificity as design platform elements provide. For example, within the Java target platform, mapping a *«Persistence»* class of a design model directly based on the JDBC API coming with the JSE API platform is much more complex than mapping it based on the JPA provided by JEE API platform.

[1]We know that adapting programming paradigm can be a compiler phenomenon, like the early C++ compiler emitting C as the intermediate code [Aho+08]. However, this strategy is not adopted by the current version of MOCCA for model transformation.

- A target platform does not have to represent a universal OOPL, but is allowed to represent a domain specific OOPL for mapping special kind of applications.

In the both previous considerations, a target platform can be used as the basis to map any kind of application modeled based on the constructs provided by the design platform by selecting suitable API or framework sub-platform. This consideration deals with the potential *domain specific* OOPL target platforms, which are not universal applicable but best suitable for mapping special kind of applications. Such target platform can be the ABAP Objects for developing mission-critical business applications and DVDL for developing *Manufacturing Execution Systems* (MES) [Dör13]. It is obvious that a modeled application with the semantics not covered by such domain specific target platforms cannot be mapped onto those platforms correctly. For example, the PIM of a 3D-application [2] can neither mapped onto ABAP Objects target platform nor onto DVDL target platform.

- A target platform accepted by MOCCA does not have to be completely identical to the original platform specification but can be fine-tuned by *adapters* and extended by in-house, ready-to-use *building blocks*.

For special OOPL target platform like ABAP Objects, some legacy non-OO language constructs but still intensively in use, can be encapsulated by adapters integrated into the target platform. The adapters are usually object-oriented wrappers, which have to be developed with the target language directly for the current version of MOCCA. The in-house, ready-to-use building blocks refer to the support of iterative modeling and model mapping. For example, a design model, which involves some kind of standard building blocks for a particular application domain, is transformed and mapped onto a target platform. The mapped result can be involved into the according target platform as ready-to-use modules. Typical example is the *account* for all kinds of applications in terms of banking management.

[2] The current MOCCA DPM is not yet equipped with the platform independent modeling facility supporting modeling 3D applications. However, with the extensible mechanism of DPM, such modeling facility can be added in the future development of MOCCA.

4.3.3. The Model Mapper Architecture

In this section, the most complicated and sophisticated component in MOCCA - the model mapper is addressed. As summarized at the end of Section 4.3.1, for each concrete implementation platform, there should be an according model mapper. However, as described in Section 3.1 and 4.3.2, the MOCCA modeling framework is completely based on object-oriented paradigm and the design models created within this modeling framework conform to the best practices of OO-technology. On the other hand, the platforms supported by MOCCA are also object-oriented. Hence, despite the diversity of target platforms, there are many common aspects among them, which lead to a hierarchical model mapper architecture illustrated in Figure 4.9.

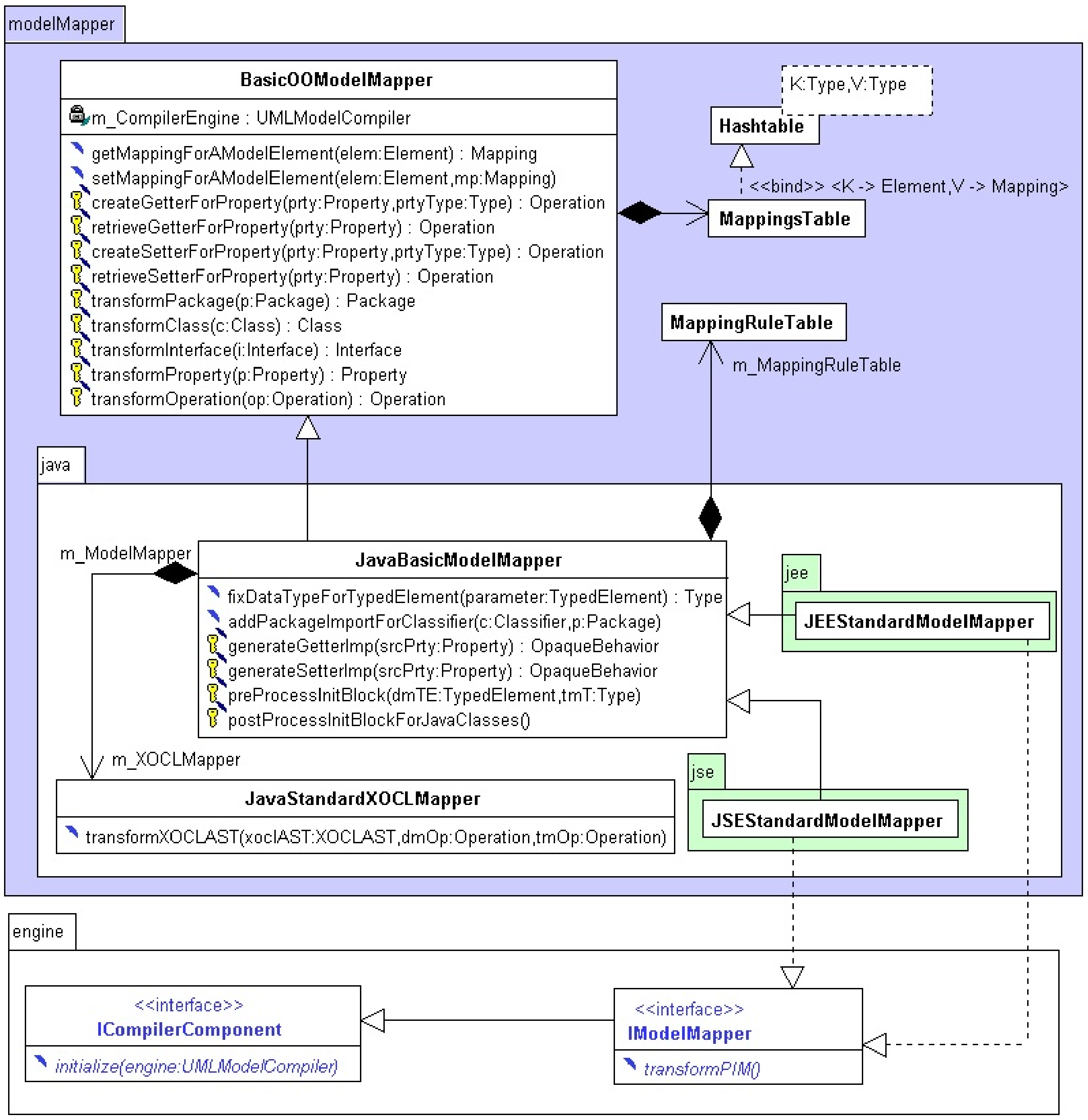

modelMapper
BasicOOModelMapper
m_CompilerEngine : UMLModelCompiler
getMappingForAModelElement(elem:Element) : Mapping
setMappingForAModelElement(elem:Element,mp:Mapping)
createGetterForProperty(prty:Property,prtyType:Type) : Operation
retrieveGetterForProperty(prty:Property) : Operation
createSetterForProperty(prty:Property,prtyType:Type) : Operation
retrieveSetterForProperty(prty:Property) : Operation
transformPackage(p:Package) : Package
transformClass(c:Class) : Class
transformInterface(i:Interface) : Interface
transformProperty(p:Property) : Property
transformOperation(op:Operation) : Operation
K:Type,V:Type
Hashtable
<<bind>> <K -> Element,V -> Mapping>
MappingsTable
MappingRuleTable
m_MappingRuleTable
java
m_ModelMapper
JavaBasicModelMapper
fixDataTypeForTypedElement(parameter:TypedElement) : Type
addPackageImportForClassifier(c:Classifier,p:Package)
generateGetterImp(srcPrty:Property) : OpaqueBehavior
generateSetterImp(srcPrty:Property) : OpaqueBehavior
preProcessInitBlock(dmTE:TypedElement,tmT:Type)
postProcessInitBlockForJavaClasses()
jee
JEEStandardModelMapper
m_XOCLMapper
JavaStandardXOCLMapper
transformXOCLAST(xoclAST:XOCLAST,dmOp:Operation,tmOp:Operation)
jse
JSEStandardModelMapper
engine
<<interface>>
ICompilerComponent
initialize(engine:UMLModelCompiler)
<<interface>>
IModelMapper
transformPIM()

Fig. 4.9.: Architecture of the model mapper component

As the root of model mapper, the *BasicOOModelMapper* takes over the general purpose mapping, which is common for all target platforms. All the platform specific mappings are delegated to the down-side model mapper(s). Each action done in a model-mapping step results in some kind of manipulation of the newly constructed target model. Hence, from very beginning on, all those actions, like *creation*, *modification*, etc., are recorded into a sub-component called *MappingsTable* for further usage. The mapping table will be addressed later in this section. As the next level of model mapping, model mapper for a concrete implementation language can be introduced. *In this dissertation, model-mappings targeted to Java programming language but on different API platforms are used to demonstrate the most essential model mapping principles. Mappings targeted to other implementation language platforms just follow the same idea.* In the Chapter *Experimental Results* of this thesis, other model mappers, which are developed in the scope of this dissertation, will be introduced and highlighted with the logic dealing with mapping issues special to those target platforms.

As modeled in Figure 4.9, the *JavaBasicModelMapper* maintains a hash-table based *MappingRuleTable*, whose mapping rules define how the model elements in design platform model can be mapped into the resources provided and abstracted in the Java target platform model. If a typed element in design model is defined using a DPM data type, the *fixDataTypeForTypedElement()* operation consults the mapping rule for that DPM data type to find the appropriate data type in Java TPM for mapping this typed element in Java target model. As aforementioned, mapping rules are defined in the external mapping configuration file and parsed into their abstract syntax (the meta model). This topic will be addressed in Section 4.3.4 in detail.

For Java model mapper, the target language is known. Hence, the *JavaStandardXOCLMapper* is connected to this level of model mapper, whose working principle will be discussed in Section 4.3.5. For now, it is to be aware that each XOCL abstract syntax tree maintained by an operation in design model will be traversed in the method *transformXOCLAST()* to generate Java code reflecting the original XOCL expression and to save it into a fresh constructed opaque behavior attached to the operation-counterpart in Java target model.

As the last level of model mapper, the API platform model mappers, such as *JSEStandardModelMapper* and *JEEStandardModelMapper*, each of which resides in its own sub-package, have enough knowledge to process the semi-finished model elements handed over by the model mapper one level before into the final model elements according to the requirements on JSE as well as JEE platform respectively.

Again, together with the overall compiler architecture depicted in Figure 4.3, it is aware that functional component like model mapper is dynamically integrated into a compilation flow by the engine via interface. The sole interface that a non-intermediate model mapper has to implement is the *IModelMapper*, which inherits the function from the *ICompilerComponent* interface. In fact, the *ICompilerComponent* interface is the base interface for all the functional interfaces. The sole operation *initialize()* injects the compiler engine into each compiler component as context, in order that the infrastructure component like *CompilerConfig*, *ModelFacility* as well as *Tracer* are accessible by functional component like model mapper. The compiler engine starts a model mapping by calling the *transformPIM()* operation via *IModelMapper* interface.

Despite the diversity of concrete target platforms, the general mapping strategy implemented by different model mappers can be high-level depicted in Alogrithm 4.4. The core-steps of a model mapping consist of the transformations in lines 5 to 9 as well as in line 15 and 21. These transformations are ubiquitous for almost all the target platforms. The others are optional. The operations prefixing with *transform* and implementing the core-transformation logic usually redefine the operations with the same signatures but defined in the intermediate model mappers along the inheritance hierarchy, e.g., the ones coming with *BasicOOModelMapper* as modeled in Figure 4.9. Both the compulsory and optional mapping steps are organized in a natural order, in which all the dependent elements in the current mapping step have been generated in the previous mapping step. For example, at the time of mapping relationships between classifiers in design model, all the counterparts of theses DM-classifiers in the target model have been generated in previous steps. This strategy is feasible, because, as already addressed in Section 4.2, at the time of model validation, a powerful symbol table recording all the model elements has been created. Hence, querying and retrieving a model element is efficient in model mapping.

Algorithm 4.4 Transform platform independent model

```
Require: Model facility containing DPM, DM as well as TPM
Ensure: Target Model (TM) is created and saved into model facility
 1: if necessary then
 2:     do some platform-specific pre-processing
 3: end if
 4:
 5: map all the packages $dmPkg \in DM$ by calling $transformPackage(dmPkg)$
 6: map all the interfaces $dmIntf \in DM$ by calling $transformInterface(dmIntf)$
 7: map all the classes $dmCls \in DM$ by calling $transformClass(dmCls)$
 8: map inheritance of each $dmCls \in DM$ by calling $transformInheritance(dmCls)$
 9: map all the properties $dmPrty \in DM$ by calling $transformProperty(dmPrty)$
10:
11: if necessary then
12:     interpret associations to obtain additional information for platform-specific processing
13: end if
14:
15: map all the operation signature $dmOp \in DM$ by calling $transformOperation(dmOp)$
16:
17: if necessary then
18:     do some platform-specific processing before behavioral mapping
19: end if
20:
21: map all the behaviors for $dmOp \in DM$ by calling $transformBehavior(dmOp)$
22:
23: if necessary then
24:     do some platform-specific post-processing
25: end if
```

As aforementioned in this section, each step in model mapping causes one or more manipulations in the constructed target model. Such manipulations are called *mapping actions*. Mapping actions in terms of one source model element are combined into one single *mapping*-object, which is recorded in the *mappings table* maintained directly by the top-level model mapper - the *BasicOOModelMapper*. The integration of mappings table into model mapper is shown in Figure 4.9. Figure 4.10 concentrates on the concepts of mapping and its mapping actions. The *MappingAction* is able to record semantic actions common among all the platforms, whereas *JEEMappingAction* enhances common mapping action with JEE-specific mapping semantics. Similarly, for each target platform, there can be specific mapping action.

The information recorded in a mapping action object consists of two parts - the model element involved and the according action done. The involved model element is refered as *resultElement* with the type of UML meta class *Element*. That means all kinds of model elements can be modified in a model mapping process. The exact semantics of a mapping action is given by assigning a mapping constant to it, which is defined in one of the *mapping constants* interfaces. Listing 4.4 shows the semantic masks, which are common among diverse target platforms. All the mapping constants are self descriptive. Hence, a further explanation for them is unnecessary.

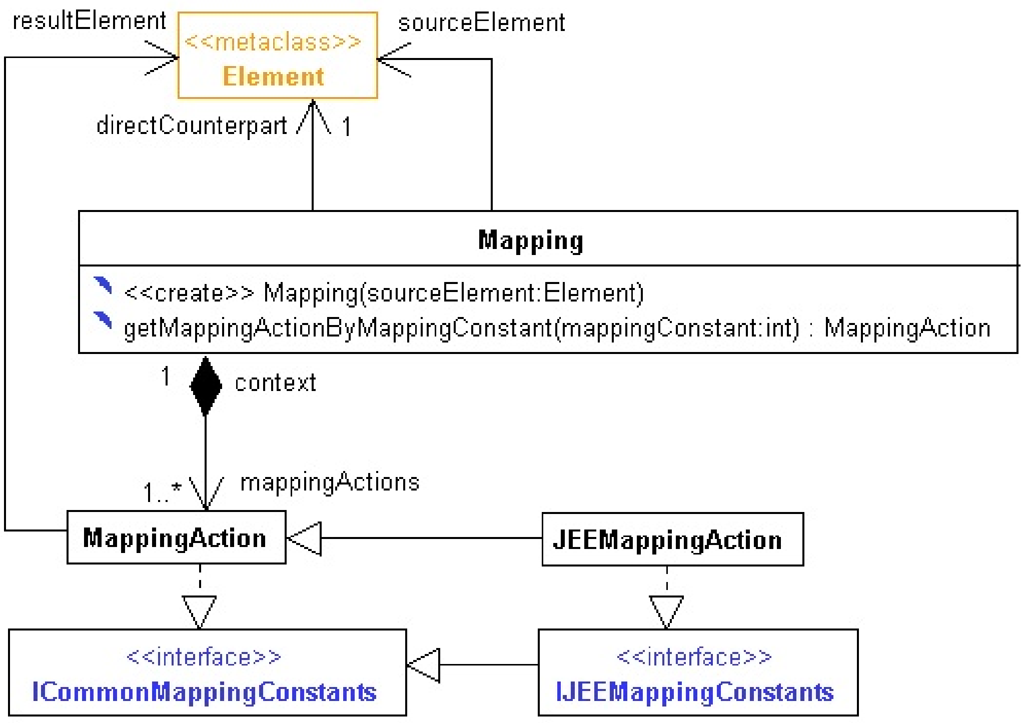


Fig. 4.10.: The structure of mapping action

```
package compiler.backEnd.modelMapper;

public interface ICommonMappingConstants
{
  int DefaultOperation   = 0;

  int PackageCreation    = 1;
  int ClassCreation      = 2;
  int InterfaceCreation = 3;

  int PropertyCreation   = 4;
  int OperationSignatureCreation = 5;

  int DataElementCreation = 6;
  int InheritanceCreation = 7;
  int ImplementationDependencyCreation = 8;
  int DecorationByStereotype = 9;
  int TaggedValueApplication = 10;
  int OperationImplementationCreation = 11;

  int PackageMoving    = 12;
  int ClassMoving      = 13;
  int OperationMoving = 14;

  int PropertyGetterCreation = 15;
  int PropertySetterCreation = 16;

  int LocalVariableCreation        = 17;
  int OperationParameterCreation = 18;
  int ModelElementRename           = 19;
}
```

Listing 4.4: Constants representing concrete mapping semantics in mapping actions

To make discussion simple, the example of model mapping shown in Figure 4.8 is simplified by highlighting only the package transformation done for the JEE-platform in Figure 4.11. On the left hand side is the source package in design model denoted as *dmPkgBO*, whereas on the right hand side are resulted packages in JEE-target model, denoted as *tmPkgBO*, *tmPkgCS*, *tmPkgRBS*, *tmPkgEntity* and *tmPkgEJB* respectively. As the stereotyped dependencies denote, the *BasicOOModelMapper* generates the *tmPkgBO* package for *dmPkgBO* package as its *direct counterpart* in JEE target model. After that a *mapping* object is created with *dmPkgBO* as its *source element* and maintains at this moment a *mapping action* object with *tmPkgBO* as its *result element* and assigned with *PackageCreation* as its mapping mask. Due to interpreting the *«BusinessObjectsLayer»* stereotype, the *JEEStandardModelMapper* generates furthermore the *tmPkgCS*, *tmPkgRBS*, *tmPkgEntity* as well as *tmPkgEJB* sub-packages and creates their according mapping action objects recording these manipulations into the same mapping, which is created by *BasicOOModelMapper* for the source package *dmPkgBO*. In this manner, all the mappings maintaining their respective mapping actions are recorded into mappings table for efficient queries in the subsequent transformation.

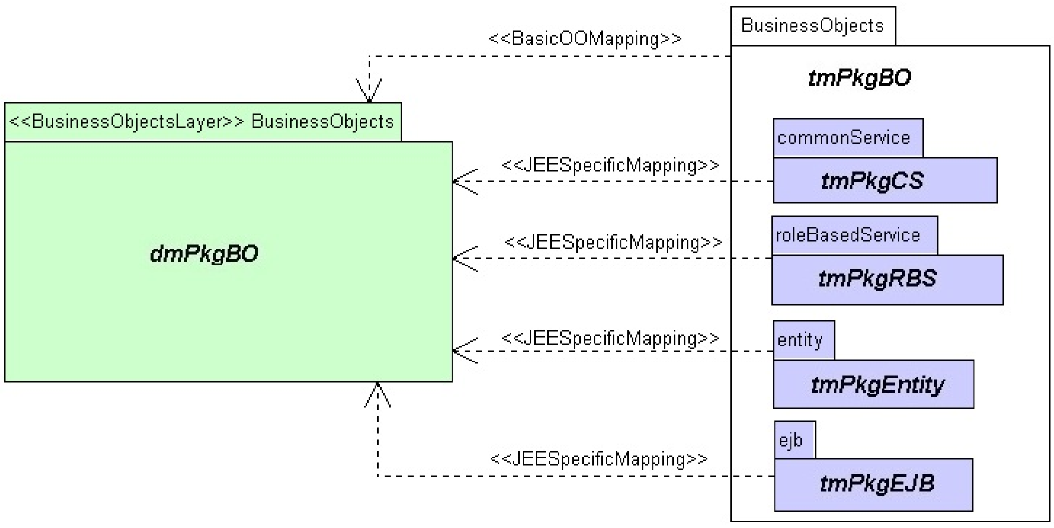


Fig. 4.11.: JEE mapping actions for transforming a package in design model

The Algorithm 4.4 illustrates the overall transformation strategy only on high-level. The concrete implementations of each transformation step in non-intermediate model mapper can be very different from platform to platform. To concentrate on essential principles, the core mapping step *transformClass()* realized in the *JEEStandardModelMapper* is used to address the general idea of implementing such a core transformation step in a concrete model mapper. Algorithm 4.5 exposes the details of this operation to some extent. The most complicated steps involved in Algorithm 4.5 are to reflect platform specific details on JEE platform. We suppose that the readers know about the underlying development issues based on JEE. The most steps in Algorithm 4.5 are self-descriptive. It is worth noting that model mapper, which is more general, hands over always a fresh created class in target model, which will be processed in the current class-transformation as shown in line 1. The following logic can be summarized as interpreting DPM stereotypes into JEE-specific transformation steps. One more thing to remind here is that due to the clarity of showing algorithm all the creations of mapping action objects and recording them into mappings table are ignored in Algorithm 4.5. In fact, for the first *if*-block in lines 2 to 7, four mapping actions representing *class movement*, *annotation adding*, *implementation dependency creation* and *operation signature creation* respectively, are generated for the source class *c* for further usage.

Algorithm 4.5 Transform design model class into JEE target model

Require: A design model class: c

Ensure: The counterpart of c in JEE target model: $tmCls$ and other related target model elements

```
 1: $tmCls \leftarrow super.transformClass(c)$
 2: if $c$ is stereotyped by «$Persistence$» then
 3:     move $tmCls$ into $entity$ package
 4:     annotate $tmCls$ with JPA @$Entity$ annotation
 5:     add implementation dependency between $tmCls$ and $Serializable$ interface
 6:     create a default constructor without parameter for $tmCls$
 7: end if
 8:
 9: if $c$ is stereotyped by «$Application$» then
10:     move $tmCls$ into $ejb$ package
11:     annotate $tmCls$ with EJB @$Startup$ annotation
12:     annotate $tmCls$ with EJB @$Singleton$ annotation
13:     annotate $tmCls$ with EJB @$LocalBean$ annotation
14:     rename $tmCls$ by suffixing $SGSB$
15: end if
16:
17: if $c$ is stereotyped by «$AdminRole$» then
18:     create EJB @$Remote$ interface in $roleBasedService$ package with naming convention
        $IAdminRole + c.name + Remote$
19:     create EJB @$Stateful$ session bean class in $ejb$ package with naming convention
        $AdminRole + c.name + SFSB$
20:     add implementation dependency between the @$Stateful$ session bean class and its
        @$Remote$ interface
21:     associate this @$Stateful$ session bean class with $tmCls$ via a property called $app$ and
        annotated by @$EJB$
22: end if
23:
24: if $c$ is stereotyped by «$BusinessRole$» then
25:     do similar transformation as for «$AdminRole$»
26: end if
27:
28: if $c$ is stereotyped by «$CommonRole$» then
29:     do similar transformation as for «$AdminRole$»
30: end if
31:
32: if $c$ is stereotyped by «$DataMapper$» then
33:     annotate $tmCls$ as EJB @$Stateless$ session bean class
34:     create EJB @$Local$ interface in the $persistenceLayer$ package with naming convention
        $I + tmCls.name + Local$
35:     add implementation dependency between $tmCls$ and its @$Local$ interface
36:     create a property $em$ with the type JPA $EntityManager$ and annotated by
        @$PersistenceContext$
37: end if
```

Algorithm 4.6 Interpret association as JPA entity relationship on JEE platform

Require: An association in design model: *dmAsso*

Ensure: Annotating the TM properties in terms of the association-ends of *dmAsso* with correct JPA annotations

```
 1:
 2: $dmRole_1 \leftarrow dmAsso.assEnd[1]$
 3: $dmRole_2 \leftarrow dmAsso.assEnd[2]$
 4: $dmPC_1 \leftarrow dmRole_1.type$
 5: $dmPC_2 \leftarrow dmRole_2.type$
 6:
 7: if $dmPC_1$ is stereotyped by «$Persistence$» $\wedge$ $dmPC_2$ is stereotyped by «$Persistence$» then
 8:     $tmPrty_1 \leftarrow dmRole_1.mapping.counterPart$
 9:     $tmPrty_2 \leftarrow dmRole_2.mapping.counterPart$
10:     if $tmPrty_1 \neq null \wedge tmPrty_2 \neq null$ then
11:         $isBidirectional = true$
12:     else
13:         $isBidirectional = false$
14:     end if
15:                                      ▷ All relationships are considered from $dmPC_1$ to $dmPC_2$
16:     if $dmRole_1.multiplicty > 1$ then      ▷ @$OneToMany$ or @$ManyToMany$ for $dmRole_1$
17:         if $dmRole_2.multiplicty > 1$ then
18:             if $isBidrectional$ then
19:                 process as @$ManyToMany$ bidirectional
20:             else
21:                 process as @$ManyToMany$ unidirectional
22:             end if
23:         else
24:             if $isBidirectional$ then            ▷ process as @$OneToMany$ bidirectional
25:                 annotate $tmPrty_2$ with @$ManyToOne$
26:                 annotate $tmPrty_1$ with @$OneToMany$
27:                 @$OneToMany.mappedBy \leftarrow tmPrty2.name$
28:             else
29:                 process as @$OneToMany$ unidirectional
30:             end if
31:         end if
32:     else                                 ▷ @$OneToOne$ or @$ManyToOne$ for $dmRole_1$
33:         if $dmRole_2.multiplicty > 1$ then
34:             if $isBidrectional$ then
35:                 process as @$ManyToOne$ bidirectional
36:             else
37:                 process as @$ManyToOne$ unidirectional
38:             end if
39:         else
40:             if $isBidrectional$ then
41:                 process as @$OneToOne$ bidirectional
42:             else
43:                 process as @$OneToOne$ unidirectional
44:             end if
45:         end if
46:     end if
47: end if
```

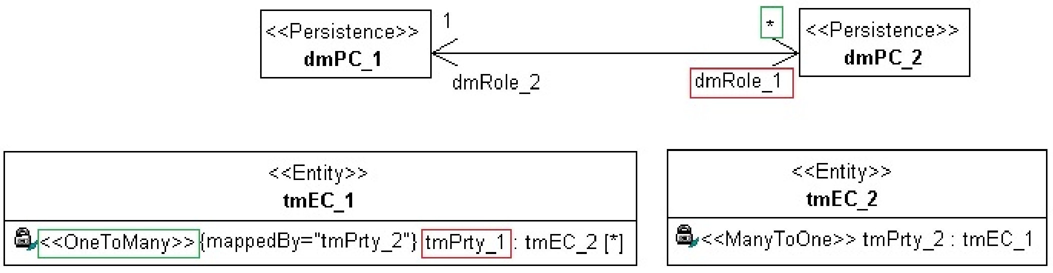


Fig. 4.12.: Interpretation of association as JPA entity relationship on JEE platform

As addressed in Section 4.2, the associations have been processed in the phase of model validation, so that their navigable association-ends have been solved into normal properties. In model transformation, these resolved properties are transformed by according *transformProperty()* operation into target model. However, for certain target platforms, associations carry more information than the relationship between classes, which can be broken down into respective properties. For example, to transform association between two *«Persistence»*-classes in DM into JPA entity relationship on JEE platform, the original association in DM has to be analyzed in detail for annotating properties in JEE target model with correct JPA annotations [Ora11b] [LB10]. The optional transformation of associations is involved in Algorithm 4.4 by the *if*-block in lines 11 to 13. Algorithm 4.6 highlights the most significant logic in the *transformAssociation()* operation implemented in *JEEStandardModelMapper*. To make discussion simple and clear, an example illustrating JPA *bidirectional one to many* entity relationship is given in Figure 4.12. Due to the same reason, only the transformation details for the case recognized as *bidirectinal one to many* are represented in lines 24 to 27 in Algorithm 4.6. Other cases just follow the similar logic.

The model elements in Figure 4.12 follow the same naming convention as in Algorithm 4.6. On the upper part of Figure 4.12, two *«Persistence»* classes in design model, denoted as *dmPC_1* and *dmPC_2* respectively, are connected by a bidirectional association. *Bidirectionality* is recognized by the both navigable association-ends, named *dmRole_1* and *dmRole_2* respectively. In model transformation, the corresponding JPA entity classes on JEE target platform are generated and denoted as *tmEC_1* and *tmEC_2* as shown on the bottom part of Figure 4.12. The *transformationProperty()* operation in JEE model mapper transforms the both *dmRole_1* and *dmRole_2* into the corresponding properties of each entity class. The rectangles with frames in red color highlight the correspondence between *dmRole_1* and *tmPrty_1*. It is to note that in target model there is no association any more. The modeling semantics of association has been dissolved into normal properties for subsequent code generation. As the next mapping step run by JEE model mapper, Algorithm 4.6 analyzes the cardinality of both *dmRole_1* and *dmRole_2* to infer the JPA *one to many* relationship. In fact, the algorithm always considers the association from the first end to the second end. If in the real model, the *dmPC_1* and *dmPC_2* change their positions, the relationship deduced by Algorithm 4.6 should be *many to one*. To recognize the bidirectionality, instead of analyzing navigability of the both association-ends directly, their generated counterparts as properties are queried via mapping objects as shown in line 8 and line 9. For non-navigable association-end, no TM-property is generated. Hence, the decision for bidirectionality can be made as shown in lines 10 to 14 in Algorithm 4.6.

As required by JPA specification [Ora11b] [LB10], for a *one to many bidirectional* relationship, not only the *@OneToMany* and *@ManyToOne* annotations have to be added to the appropriate properties in entity classes, but also the *mappedBy* attribute must be assigned with the property, which represents a possible *foreign key* reference in the underlying database schema. This is done as shown in line 27 in Algorithm 4.6 and represented in the curly braces in class *tmEC_1* of Figure 4.12.

4.3.4. The Mapping Configuration and Its Metamodel

Recall from the general architecture of model mapper illustrated in Figure 4.9 on page 91, the *JavaBasicModelMapper* (the same is applied to other implementation language specific intermediate model mapper) maintains a hash-table based *MappingRuleTable*, whose mapping rules define how the model elements in design platform model can be mapped into the resources provided and abstracted in the Java target platform model. As discussed in Section 3.3, the model elements in design platform model can be classified in two distinct groups. The profile and its stereotypes are used to mark a design model element with additional semantics that can be interpreted by model mapper into resource combination on a target platform to realize the identical semantics added by such stereotypes. In previous Section, processing of this group of DPM model elements has been addressed. The other group of DPM model elements are XOCL data types and their features. Data types are primarily used to define UML typed elements such as property, operation parameter, local variable, as well as to specify general classifier in an inheritance and supplier in a dependency. Features, namely, the properties and operations defined in data types serve as the steppingstone in XOCL expressions to model behavior compactly. As already known, the XOCL extends the open standard based OCL with extra types and features. The original OCL types and their pre-defined operations are completely preserved. Hence, the mappings for platform data types follow a general strategy consisting of:

- For both original OCL types and extended XOCL types, mapping rules are defined in mapping configuration file to map them into data types on a concrete target platform.
- For OCL predefined operations, the implementation language specific XOCL mapper is able to map them into target language code. Because all the OCL predefined operations are documented in literature like [WK03] and [OMG10a], the exact semantics of OCL predefined operations are well known for a specific XOCL mapper targeted to a concrete platform.
- For XOCL features defined in XOCL types, namely properties and operations, mapping rules are needed to represent or implement them by using the combination of resources provided on a target platform.

The first and third point of the above mapping strategy are discussed in this section by examples, whereas the second point will be addressed in the next section in detail. As mentioned several times before, the mapping rules are defined in a *mapping configuration file* in XML format. Listing 4.5 shows the general structure of a mapping configuration file. It is supposed that readers of this thesis are familiar with XML. Otherwise, introduction about XML can be found in [FQA12]. As shown in Listing 4.5, a dedicated name space indicates that this XML-based mapping configuration file is created in the scope of my PHD-study. Furthermore, an XML schema, namely, *mappingConfig.xsd* is also defined to validate the elements appeared in this XML-file and delivered with MOCCA together. The complete usage and valid semantics of the mapping configuration file can be retrieved by reading the XSD file. Here a brief summary of the most significant elements are provided. The mapping configuration consists of three parts. The *<targetPlatform>* element can be used only once and is followed by the composite element *<dpmMapping>*, in which arbitrary number of sub-elements can be defined. The meaning of these sub-elements ought to be self-descriptive and will be addressed by examples in the subsequent discussion. The *<dmMapping>* is reserved for possible direct mappings between model elements in design model and their implementation on a target platform directly. For the time being, this element is not used.

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <mappingConfig xmlns="http://www.tu-freiberg.de/Promotion/Liang"
3                  xmlns:xsi ="http://www.w3.org/2001/XMLSchema-instance"
4                  xsi:schemaLocation="http://www.tu-freiberg.de/Promotion/Liang
                       mappingConfig.xsd" >
5
6    <targetPlatform>JSE</targetPlatform>
7    <dpmMapping>
8      <dpmTypeMappingRule>
9       </dpmTypeMappingRule>
10         ...
11      <dpmEventMappingRule>
12      </dpmEventMappingRule>
13        ...
14      <dpmPropertyMappingRule>
15      </dpmPropertyMappingRule>
16        ...
17      <dpmOperationMappingRule>
18      </dpmOperationMappingRule>
19   </dpmMapping>
20
21   <dmMapping>
22   </dmMapping>
23  </mappingConfig>
```

Listing 4.5: The general structure of a mapping configuration file

As the *<targetPlatform>* element in line 6 in Listing 4.5 indicates, in this section, mapping rules targeted to Java Standard Edition (JSE) are selected to illustrate typical usages of mapping rules. Mapping rules for other possible target platforms just follow the similar idea. Listing 4.6 shows both mapping rules for the OCL primitive type *Integer* and the XOCL GUI-Layout type *SplitContainer* respectively. Each mapping rule for DPM-types has a *<sourceType>* element, to which the current mapping rule is applied. The type surrounded by this element is also the *key*-object in the mapping rule table. Following the *<sourceType>* element comes one of the elements, which define the counterpart on a target platform on the one hand and the semantics applied to this counterpart on the other hand. For example, to map the OCL *Integer* on JSE, the Java *int* built-in type can be used to simply substitute the source type. This substitution semantics is defined by the element *<typeSubstitution>* and its attribute *as* gives the mapping rule parser the hint, how to interpret the JSE *int* built-in type. The extra attribute *isInit* in the *<typeSubstitution>* of the mapping rule for *SplitContainer* gives model mapper a necessary hint that the type on target platform has to be initialized before using it, which usually leads to platform specific post processing as shown in Algorithm 4.4. It is to note that for both source type and target type the qualified names have to be given in a type mapping rule.

```
 <dpmTypeMappingRule>
      <sourceType>OCL.Primitives.Integer</sourceType>
      <typeSubstitution as="DataType">java.lang.int</typeSubstitution>
 </dpmTypeMappingRule>

 <dpmTypeMappingRule>
    <sourceType>XOCL.GUI.SplitContainer</sourceType>
    <typeSubstitution as="Class" isInit="true">javax.swing.JSplitPane</
         typeSubstitution>
  </dpmTypeMappingRule>
```

Listing 4.6: Mapping rules for Integer and SplitContainer in design platform model

The mapping rule in Listing 4.7 illustrates the usage of <*typeDeclaration*> and <*typeDefinition*> elements. This mapping rule defines that the Java *List<T>* interface is used to declare a typed element like property, but initialized, for example, in constructor, by the Java *ArrayList<T>* class, for the OCL collection type *OrderedSet*. Listing 4.3 on page 86 shows a possible result if the above rule is applied.

```
<dpmTypeMappingRule>
    <sourceType>OCL.Collections.OrderedSet</sourceType>
    <typeDeclaration as="Interface" isInit="false">java.util.List</typeDeclaration>
    <typeDefinition as="Class" isInit="true">java.util.ArrayList</typeDefinition>
  </dpmTypeMappingRule>
```

Listing 4.7: Mapping rule for the collection type OrderedSet in design platform model

```
<dpmEventMappingRule>
    <sourceEvent>XOCL.GUI.Event.ClickEvent</sourceEvent>
    <eventTypeHandling>
      <tpmEventName>actionPerformed</tpmEventName>
      <tpmEventOwner as="Interface">java.awt.event.ActionListener</tpmEventOwner>
    </eventTypeHandling>
  </dpmEventMappingRule>
```

Listing 4.8: Mapping rule for the platform independent ClickEvent into Java

Recall from Section 3.3.3, the XOCL event-expression is introduced to model dynamic subscription part of event handling in a clear and platform independent manner. This mechanism supports flexible declaration of event handling methods. Specifically, their names do not need to be pre-coded. However, to map XOCL event-expressions into certain target platform, such as JSE, difficulty occurs due to the peculiarity of the underlying Java event handling framework [HC08], which can be summarized as:

- In JSE, a single event on an event source cannot be identified explicitly as in C# via *event* keyword.
- In JSE, event handling methods have to be declared in *XXXListener*-interfaces and connect to events via method named *addXXXListener()*. Only methods registered in this way can be used in the dynamic publishing phase of event handling.

As solution for the first point above, an intuitive candidate may be a Java event object, e.g., *WindowEvent*, *MouseEvent*, etc. However, they are similar to their corresponding listener interfaces, which group several related events together. It requires an extra effort to select a single event of such an event-collection. As result of the careful analysis discussed in [LS10], it is possible to adopt the *method name* defined in the event listeners to identify a single event. If an event-source can register several event listeners, the method names in this set of listeners classify the events exactly.

Following this idea the <*dpmEventMappingRule*> element is created to define the correspondence between DPM event types and their counterparts in a target platform. Listing 4.8 shows how this rule can be used to map *ClickEvent* onto JSE platform. The JSE model mapper knows how to use information provided in this mapping rule to generate correct Java code.

To overcome the second problem above, which concerns connecting the event handler with the event, a Java *anonymous class* [HC08] can be created as a bridge between the fixed method-name of a Java event-handling method and the free chosen name of the event handler in the design model. Listing 4.9 shows the relationship between an XOCL event-expression and its implementation on JSE platform. The Java code is generated automatically by the *JavaStandardXOCLMapper* to be addressed in the next section. It is to note that the appropriate listener interface or adapter class to declare the anonymous class is also given in the event mapping rule by the *<tpmEventOwner>* sub-element shown in line 5 in Listing 4.8.

```
begin
  event: self.view.button1.click ~ self.button1_click;
end

this.getView().getButton1.addActionListener(
        new ActionListener(){
                @Override
                public void actionPerformed(ActionEvent e){
                      button1_click(e);
                }
        }
);
```

Listing 4.9: An XOCL event expression and its corresponding Java implementation via anonymous class

```
  <dpmPropertyMappingRule>
      <sourceProperty>XOCL.GUI.SplitContainer.isHorizontal</sourceProperty>
      <propertySubstitution>
        <tpmPropertyName>orientation</tpmPropertyName>
        <tpmPropertyGetter>getOrientation</tpmPropertyGetter>
        <tpmPropertySetter>setOrientation</tpmPropertySetter>
        <isDirectAssigned>false</isDirectAssigned>
        <dpmPV2tpmPV>
          <dpmPV>false</dpmPV>
          <tpmPV>javax.swing.JSplitPane.VERTICAL_SPLIT</tpmPV>
        </dpmPV2tpmPV>
        <dpmPV2tpmPV>
          <dpmPV>true</dpmPV>
          <tpmPV>javax.swing.JSplitPane.HORIZONTAL_SPLIT</tpmPV>
        </dpmPV2tpmPV>
      </propertySubstitution>
   </dpmPropertyMappingRule>
```

Listing 4.10: Mapping rule for the property isHorizontal defined in XOCL GUI–Layout type SplitContainer

```
begin
  update self.splitPane.isHorizontal = false;
end

this.splitPane.setOrientation(javax.swing.JSplitPane.VERTICAL_SPLIT);
```

Listing 4.11: Mapping a property of XOCL type into Java

To map properties of an XOCL type, the *<dpmPropertyMappingRule>* element can be used. In most situations, there must be a counterpart for such a property on the target platform, but the representation on target platform can be different than in design model. The sub-elements in *<dpmPropertyMappingRule>* are provided to smooth such differences. Listing 4.10 illustrates a mapping rule, which defines how to map the *isHorizontal* property of *SplitContainer* into appropriate setting of the *orientation* property in Java *JSplitPane* class. Listing 4.11 shows the mapping result.

Mapping XOCL library operations encounters similar situations to mapping properties. For most situations, there are direct counterparts on a target platform to substitute XOCL-operations. If not the case, additional elements of the *<dpmOperationMappingRule>* can be used to smooth the differences, just like for mapping properties. The most complicated situation for mapping XOCL operation is addressed by an example in this section in detail.

Figure 4.13 shows a piece of design model, which represents the model part of the MVC architecture. The class *Module* represents a subject provided in a course of a university, whereas the class *ModuleGroup* groups the coherent modules as an examination unit. A module group

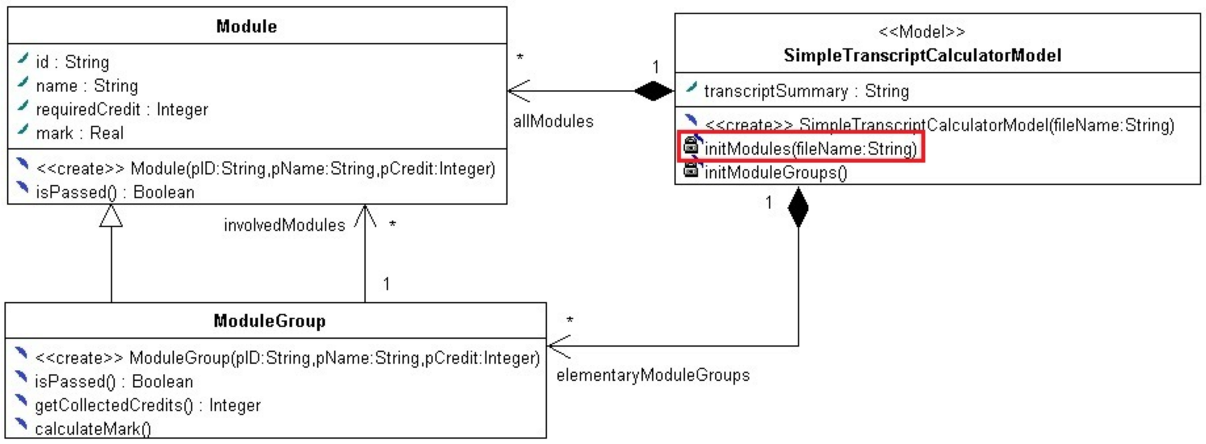

Module
id : String
name : String
requiredCredit : Integer
mark : Real
<<create>> Module(pID:String,pName:String,pCredit:Integer)
isPassed() : Boolean
*
1
allModules
<<Model>>
SimpleTranscriptCalculatorModel
transcriptSummary : String
<<create>> SimpleTranscriptCalculatorModel(fileName:String)
initModules(fileName:String)
initModuleGroups()
involvedModules
*
1
1
ModuleGroup
<<create>> ModuleGroup(pID:String,pName:String,pCredit:Integer)
isPassed() : Boolean
getCollectedCredits() : Integer
calculateMark()
*
elementaryModuleGroups

Fig. 4.13.: The Model-part of an MVC-based design model

can be nested, recognized by the combination of inheritance and association between *Module* and *ModuleGroup*. The modules are created and maintained by teaching staff and saved in a *text file*, which can be in any possible format. In this simple example, a space-separated plain text file is supposed. At the time of initializing system, the *initModules()* operation highlighted by a rectangle, whose frame is in red color, fills in the association-end *allModules* with fresh created *module*-instances by reading all the modules recorded in a file line by line. This semantics is clear at the time of modeling. Hence, a high-level XOCL operation can be declared as shown in Listing 4.12.

```
package DesignPlatformModel::XOCL::IO

context InputStream
def: readCollectionFromTextFile(fileName: String): Collection(OclAny);
```

Listing 4.12: Declaration of an XOCL library operation using OCL–like syntax

```
begin
    update self.allModules = InputStream::readCollectionFromTextFile(fileName);
end
```

Listing 4.13: A concrete usage of XOCL readCollectionFromTextFile() operation

The expression in Listing 4.12 is considered as *OCL-like*. Because the original OCL-*def* expression is used to declare a query operation in the underlying model. There is "="-part after operation signature, which states the body expression. However, the expression is thought to be convenient to declare an operation in a platform independent way. That means, the *readCollectionFromTextFile()* operation belongs to XOCL type *InputStream*, which is in the *IO* sub-package of XOCL, the input parameter is a file name and the operation brings back a collection, both of whose raw type and involved type are not yet determined. This XOCL operation can be used to specify the behavior of the *initModules()* operation as modeled in Listing 4.13. Now, the concrete return type of the *readCollectionFromTextFile()* operation can be deduced by the data type of *allModules*, which is an *OrderedSet* of *Modules*.

It is clear that in JSE there is no direct counterpart for *readCollectionFromTextFile()* operation. Furthermore, there is even no operation that can be smoothed to emulate this operation. To map such an operation onto a target platform, there are generally two possibilities:

1. Model a wrapper class with the counterpart operation in another design model and transform this model into the required target language. After that, the generated wrapper class

and its operation(s) can be adopted in the *Extended Adapter API Model*, which is a submodel in target platform model already introduced in Section 3.2. From this moment on, a simple operation mapping rule can be defined between the complex XOCL operation and its adapter counterpart in target platform model. This strategy is referred as *iterative modeling* already mentioned in Section 4.3.2.

2. The semantics of such an operation in XOCL is known meanwhile the resources provided on a target platform are also known. Thus, this operation can be realized by the combination of target platform resources. The combination can be represented in the form of code template in terms of target platform.

```
<dpmOperationMappingRule>
  <sourceOperation>
        XOCL.IO.InputStream.readCollectionFromTextFile(DesignPlatformModel.OCL.
             Primitives.String)
  </sourceOperation>

  <operationSubstitution>
     <useTPMcodeTemplate>true</useTPMcodeTemplate>
      <tpmCodeTemplate>
           try{
             java.io.BufferedReader br = new java.io.BufferedReader(new java.io.
                  FileReader( %Param0% ));
             String line ="";
             while( (line = br.readLine())!=null){
              if(!line.isEmpty()){
                 String[] lineElem = line.split(" ");
                 if(lineElem.length == %Param1% ){
                   %Param2% obj = new %Param2%( %Param3% );
                   %Param4%.add(obj);
                    }
                 }
             }
           }catch(java.io.IOException e)
           {}
      </tpmCodeTemplate>
      <tpmCodeTemplateParameter>%Param0%</tpmCodeTemplateParameter>
      <tpmCodeTemplateParameter>%Param1%</tpmCodeTemplateParameter>
      <tpmCodeTemplateParameter>%Param2%</tpmCodeTemplateParameter>
      <tpmCodeTemplateParameter>%Param3%</tpmCodeTemplateParameter>
      <tpmCodeTemplateParameter>%Param4%</tpmCodeTemplateParameter>
  </operationSubstitution>
</dpmOperationMappingRule>
```

Listing 4.14: Mapping rule for XOCL readCollectionFromTextFile() operation onto JSE platform

Listing 4.14 shows the sophisticated mapping rule for the *readCollectionFromTextFile()* operation onto JSE platform. The most complicated part in this mapping rule is covered by the template code in Java with required template parameters, which will be replaced by the concrete contents in the process of XOCL to Java mapping.

Up to this point, the fundamental syntax and semantics of the XML-based mapping configuration file have been introduced by examples. What is not yet addressed concerns the parsing of those mapping rules and their internal representation in the mapping rule table. Recall from the UML metamodel discussed in Section 2.1.2 and the abstract syntax of XOCL addressed in Section 3.4, the mapping rules are internally represented by their metamodel instances created in the process of parsing the underlying mapping configuration file. Figure 4.14 shows the class hierarchy of the metamodel of mapping rules. As aforementioned, the *MappingRuleTable* is a hash table, which supports efficient query of mapping rule defined for a single model element. The *MappingRule* has a *sourceElement* and maintains one or many *MappingRuleItem*s, each of which refers to a target platform element as the mapping result. The diverse concrete mapping rule items have their own mapping semantics and stores extra information for smoothing the differences between DPM and TPM model elements. The sequence diagram in Figure 4.15 illustrates the process of parsing mapping configuration file and establishing the mapping rule table.

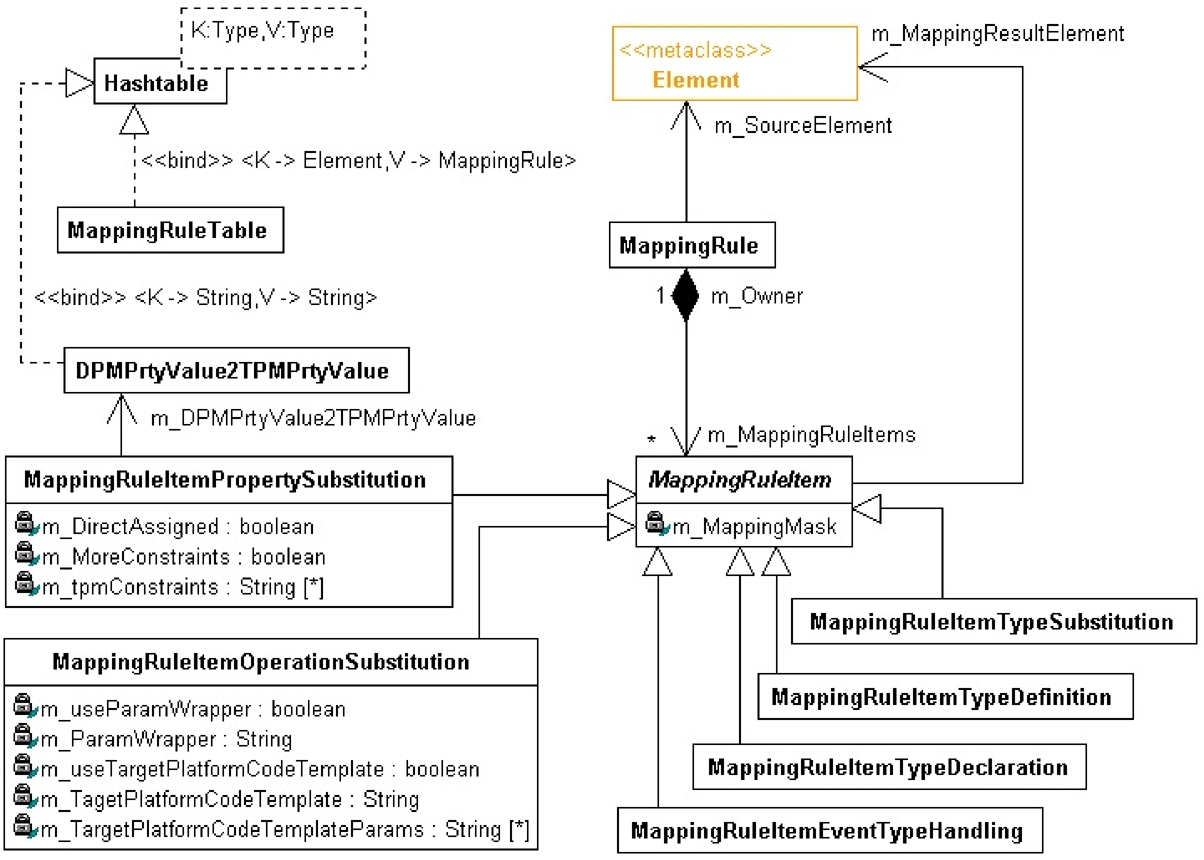

K:Type,V:Type
Hashtable
<<bind>> <K -> Element,V -> MappingRule>
MappingRuleTable
<<bind>> <K -> String,V -> String>
DPMPrtyValue2TPMPrtyValue
m_DPMPrtyValue2TPMPrtyValue
MappingRuleItemPropertySubstitution
m_DirectAssigned : boolean
m_MoreConstraints : boolean
m_tpmConstraints : String [*]
MappingRuleItemOperationSubstitution
m_useParamWrapper : boolean
m_ParamWrapper : String
m_useTargetPlatformCodeTemplate : boolean
m_TagetPlatformCodeTemplate : String
m_TargetPlatformCodeTemplateParams : String [*]
<<metaclass>>
Element
m_MappingResultElement
m_SourceElement
MappingRule
1
m_Owner
*
m_MappingRuleItems
MappingRuleItem
m_MappingMask
MappingRuleItemTypeSubstitution
MappingRuleItemTypeDefinition
MappingRuleItemTypeDeclaration
MappingRuleItemEventTypeHandling

Fig. 4.14.: The metamodel of mapping rules

The process begins with pressing the menu item or toolbar button for transforming the PIM, which triggers the execution of the corresponding event handling method defined in the *Controller*-class. First of all, the existence of a model mapper is checked. If the model mapper is not installed, the entire optional combined fragment will be executed. This occurs usually for the first time usage after a fresh bootstrap of MOCCA. Otherwise, the *transformPIM()* operation is invoked immediately. The installation of an appropriate model mapper for a special target platform involves the initialization of the mapping rule table storing mapping rules for that platform. As shown in Figure 4.15, after instantiating the *JSEStandardModelMapper*, its *loadMappingRules()* method is called to parse the mapping configuration file for JSE platform. The actual parsing is done by a *MappingRuleParser* object, which is a DOM XML parser. As an additional guarantee of establishing mapping rule table, an auxiliary component named *XXXStandardMappingRuleInitializer*, with XXX a platform indicator like *JSE* or *JEE*, will be switched on to take over the initialization process if the previous parsing failed. The mapping rule initializer fills in the mapping rule table with default mapping rules based on the general knowledge about the underlying target platform.

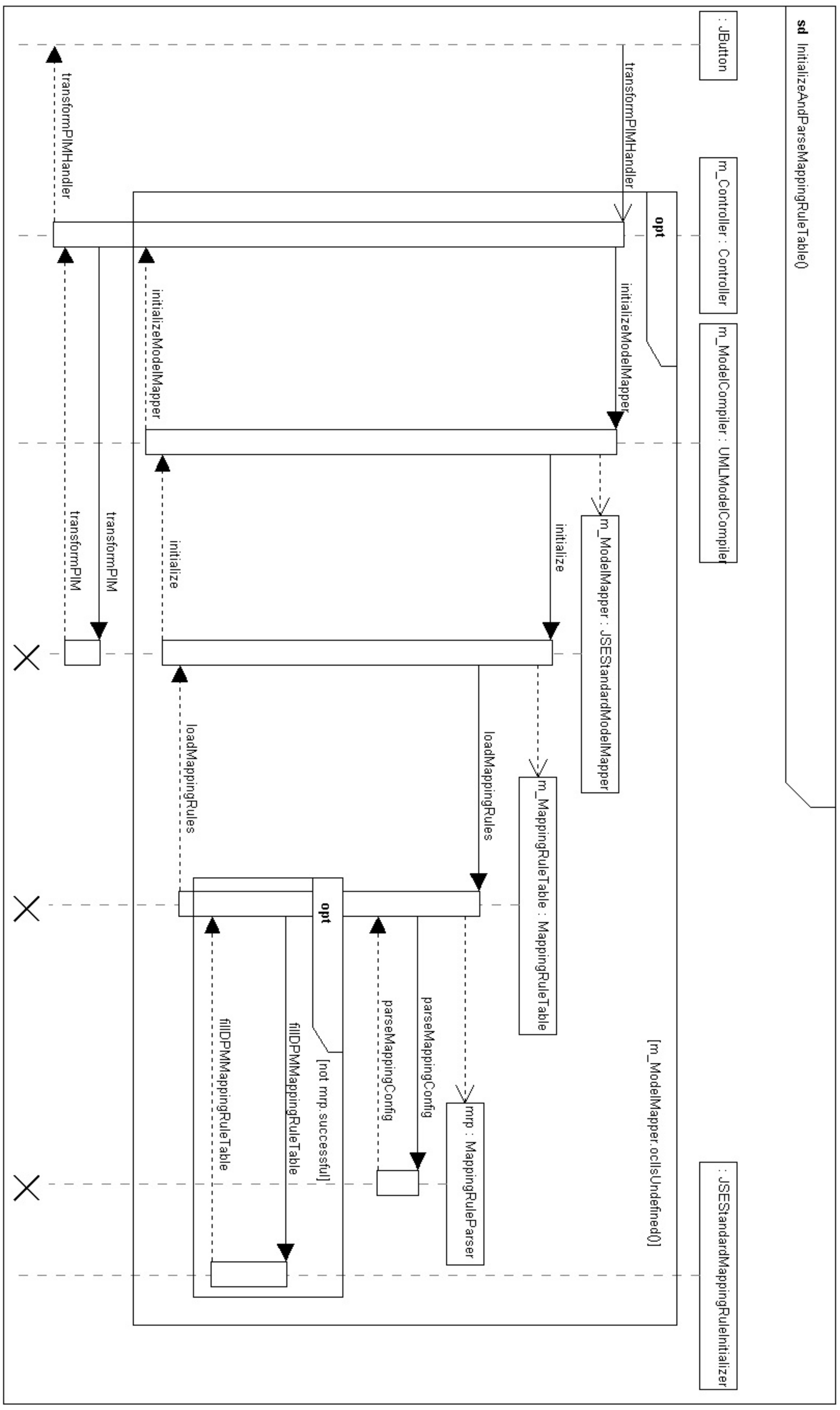


Fig. 4.15: The process of parsing mapping configuration file and establishing the mapping rule table

Algorithm 4.7 Fix data type of a typed element

Require: A typed element defined in design model: te

Ensure: The correct data type of te mapped on target platform: $tmFinalDT$

```
 1: $dmType \leftarrow te.type$
 2:
 3: if $dmType$ is $XOCLCollection$ then
 4:     $dmColl \leftarrow dmType.rawTypeDecl$
 5:     $dmContent \leftarrow dmType.contentType$
 6: else if $dmType$ is $XOCLHashTable$ then
 7:     $dmColl \leftarrow dmType.rawTypeDecl$
 8:     $dmHTKey \leftarrow dmType.keyType$
 9:     $dmContent \leftarrow dmType.valueType$
10: else
11:     $dmContent \leftarrow dmType$
12:     if $te$ is $Property \vee te$ is $Parameter$ then
13:         if $te.multiplicity > 1$ then
14:             $dmColl \in \{Set, OrderedSet, Bag, Sequence\}$ by checking $te.isOrdered$ and $te.isUnique$
                marks as shown in Table 2.1
15:         end if
16:     end if
17: end if
18:
19: if $dmColl \neq null$ then
20:     $tmCollDecl, tmCollDef \leftarrow this.mappingRuleTable.getTypeSubstitutionTPMType(dmColl)$
21:     if $tmCollDecl = null$ then
22:         $tmCollDecl \leftarrow this.mappingRuleTable.getTypeDeclarationTPMType(dmColl)$
23:         $tmCollDef \leftarrow this.mappingRuleTable.getTypeDefinitionTPMType(dmColl)$
24:     end if
25:     $init \leftarrow true$
26: end if
27:
28: if $dmContent \neq null$ then
29:     if $dmContent$ is data type on design platform then
30:         retrieve $tmContent$ from the mapping rule for $dmContent$
31:     else
32:         retrieve $tmContent$ from the direct counterpart of the $mapping$ created for $dmContent$
            at the time of model transformation
33:     end if
34: end if
35:
36: if $dmHTKey \neq null$ then
37:     do similar type mapping as for $dmContent$
38: end if
39:
40: construct $tmFinalDT$ from compulsory $tmContent$ and optional $tmCollDecl$, $tmCollDef$
    and $tmHTKey$
41:
42: if $init$ then
43:     pre-process the pair $(te, tmFinalDT)$ as init-block
44: end if
```

The *fixDataTypeForTypedElement()* operation defined in class *JavaBasicModelMapper* as shown in Figure 4.9 on page 91 is strongly dependent on both the *MappingsTable* and *MappingRuleTable*. Algorithm 4.7 exploits the essential working principle of this operation. In lines 20, 22 and 23 of Algorithm 4.7, the concrete query operations of mapping rule table are listed, whereas in line 30 only pseudo instruction is used for simplicity. Line 32 illustrates that the original data type of a typed element can be a design class, whose TM-counterpart is recorded in the *mapping*-object for that class instead of in mapping rule table.

4.3.5. Traversal of XOCL Abstract Syntax Tree For Behavioral Mapping

In the previous sections, the structural mapping for model elements in the class diagram of a design model into their counterparts on a target platform has been addressed. In this section, the behavioral mapping for operation specifications in design model will be discussed. The problem itself can be traced back to classical compiler design [Aho+08]. Hence, this section concentrates on concrete source-representation as well as target code issue rather than mathematical issues of compiler design. Another important objective is to provide a general guide for developing this back-end component, namely, the *XOCL mapper* for a concrete target language.

Recall from Section 3.4 and Section 4.2, the means used to model behavior in the context of MOCCA is XOCL. After model parsing and validation, all the XOCL expressions saved in opaque behaviors attached to their owning operations have been parsed, type-checked as well as transformed into their abstract syntax trees rooting in *XOCLAST* objects, which replace the original opaque behaviors for the subsequent *XOCL to target code* mapping. To make discussion simple and clear, an example is used to review important working principles of model mapper and moreover, to explore the most significant capabilities required by an XOCL Mapper.

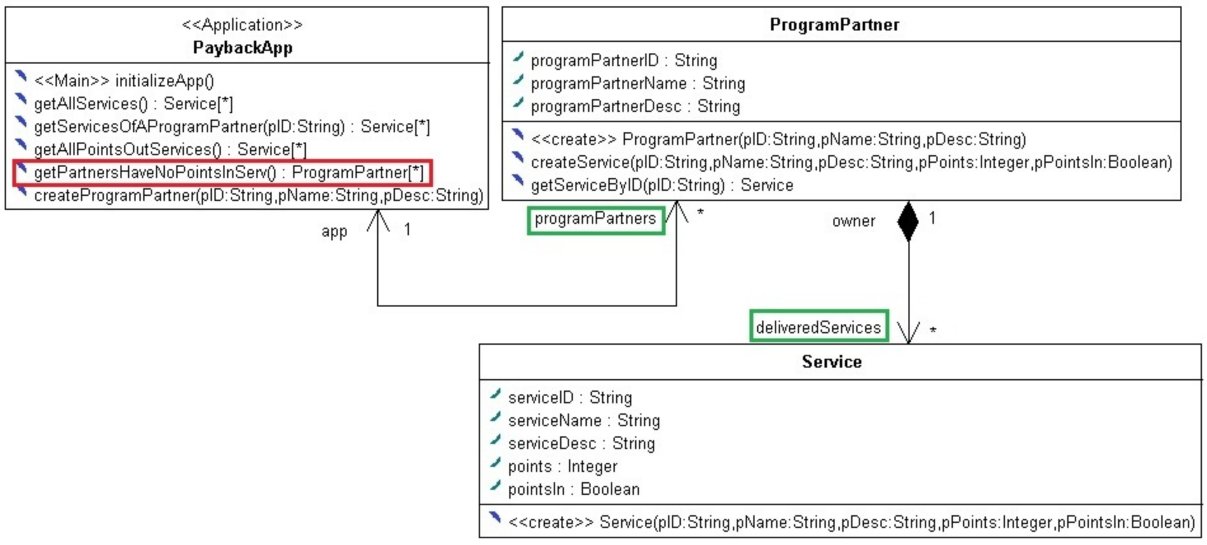

<<Application>>
PaybackApp
<<Main>> initializeApp()
getAllServices() : Service[*]
getServicesOfAProgramPartner(pID:String) : Service[*]
getAllPointsOutServices() : Service[*]
getPartnersHaveNoPointsInServ() : ProgramPartner[*]
createProgramPartner(pID:String,pName:String,pDesc:String)
ProgramPartner
programPartnerID : String
programPartnerName : String
programPartnerDesc : String
<<create>> ProgramPartner(pID:String,pName:String,pDesc:String)
createService(pID:String,pName:String,pDesc:String,pPoints:Integer,pPointsIn:Boolean)
getServiceByID(pID:String) : Service
app
1
programPartners
*
owner
1
deliveredServices
*
Service
serviceID : String
serviceName : String
serviceDesc : String
points : Integer
pointsIn : Boolean
<<create>> Service(pID:String,pName:String,pDesc:String,pPoints:Integer,pPointsIn:Boolean)

Fig. 4.16.: Design model: a simple design model with operations exposing typical mapping issues for XOCL

```
body:  self.programPartners->select(deliveredServices
                                     -> forAll(not pointsIn ))
```

Listing 4.15: Design model: XOCL expression specifying the query operation getPartnersHaveNoPointsInServ() in Figure 4.16

Figure 4.16 shows a simple design model for a payback system, in which only the joint-in program partners and their delivered services are modeled. Several operations with self-descriptive

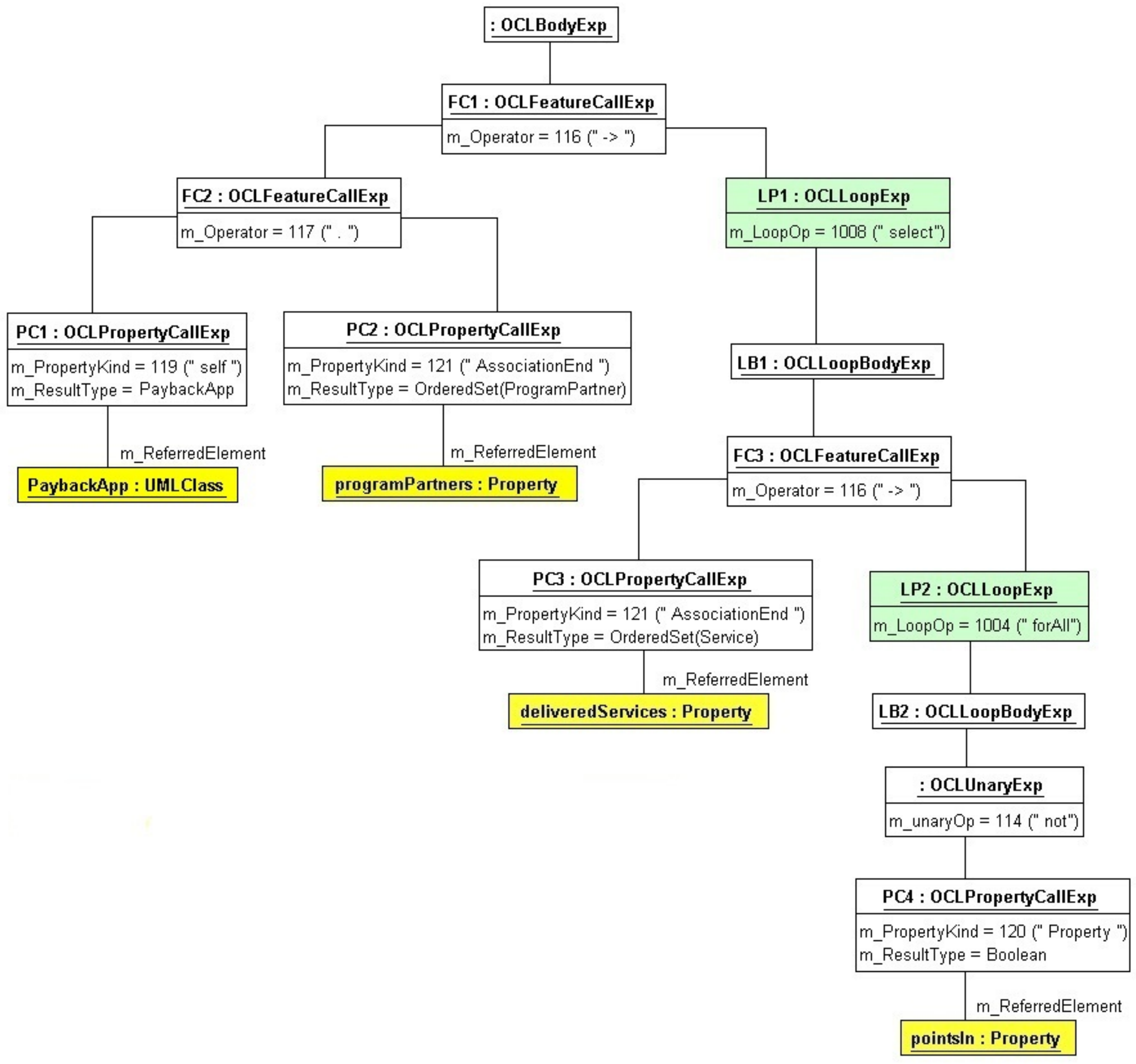

: OCLBodyExp
FC1 : OCLFeatureCallExp
m_Operator = 116 (" -> ")
FC2 : OCLFeatureCallExp
m_Operator = 117 (" . ")
LP1 : OCLLoopExp
m_LoopOp = 1008 (" select")
PC1 : OCLPropertyCallExp
m_PropertyKind = 119 (" self ")
m_ResultType = PaybackApp
PC2 : OCLPropertyCallExp
m_PropertyKind = 121 (" AssociationEnd ")
m_ResultType = OrderedSet(ProgramPartner)
LB1 : OCLLoopBodyExp
m_ReferredElement
PaybackApp : UMLClass
programPartners : Property
FC3 : OCLFeatureCallExp
m_Operator = 116 (" -> ")
PC3 : OCLPropertyCallExp
m_PropertyKind = 121 (" AssociationEnd ")
m_ResultType = OrderedSet(Service)
LP2 : OCLLoopExp
m_LoopOp = 1004 (" forAll")
deliveredServices : Property
LB2 : OCLLoopBodyExp
: OCLUnaryExp
m_unaryOp = 114 (" not")
PC4 : OCLPropertyCallExp
m_PropertyKind = 120 (" Property ")
m_ResultType = Boolean
pointsIn : Property

Fig. 4.17.: Design model: the abstract syntax tree generated for the XOCL expression in Listing 4.15

signatures are also defined in this model. The operation highlighted by a red rectangle returns all the program partners, whose all delivered services cannot accumulate points. This semantics can be specified clearly and compactly in an XOCL expression given in Listing 4.15. After model validation, the exact abstract syntax tree as shown in Figure 4.17 in the form of UML object diagram is generated. In an AST like this, each node has concrete semantics and stores important information, such as *resulted data type* and *referred model element*. Concrete to this example, the both association-ends denoted by rectangles with green frame color in Figure 4.16 and used as *callers* for *select()* and *forAll()* operations in Listing 4.15 are represented in the abstract syntax tree in Figure 4.17 as AST-nodes named *PC2* and *PC3*, each of which is an *OCLPropertyCallExp* and references to the actual *property* (association-ends in both cases) as well as storing the result data type by evaluating the property.

After *model transformation*, in this case, a *DM to JSE target model* transformation, the *JSE-StandardModelMapper* constructs the target model on JSE platform as illustrated in Figure 4.18. With discussions in the both previous sections in mind, the essential structural mappings can be summarized as:

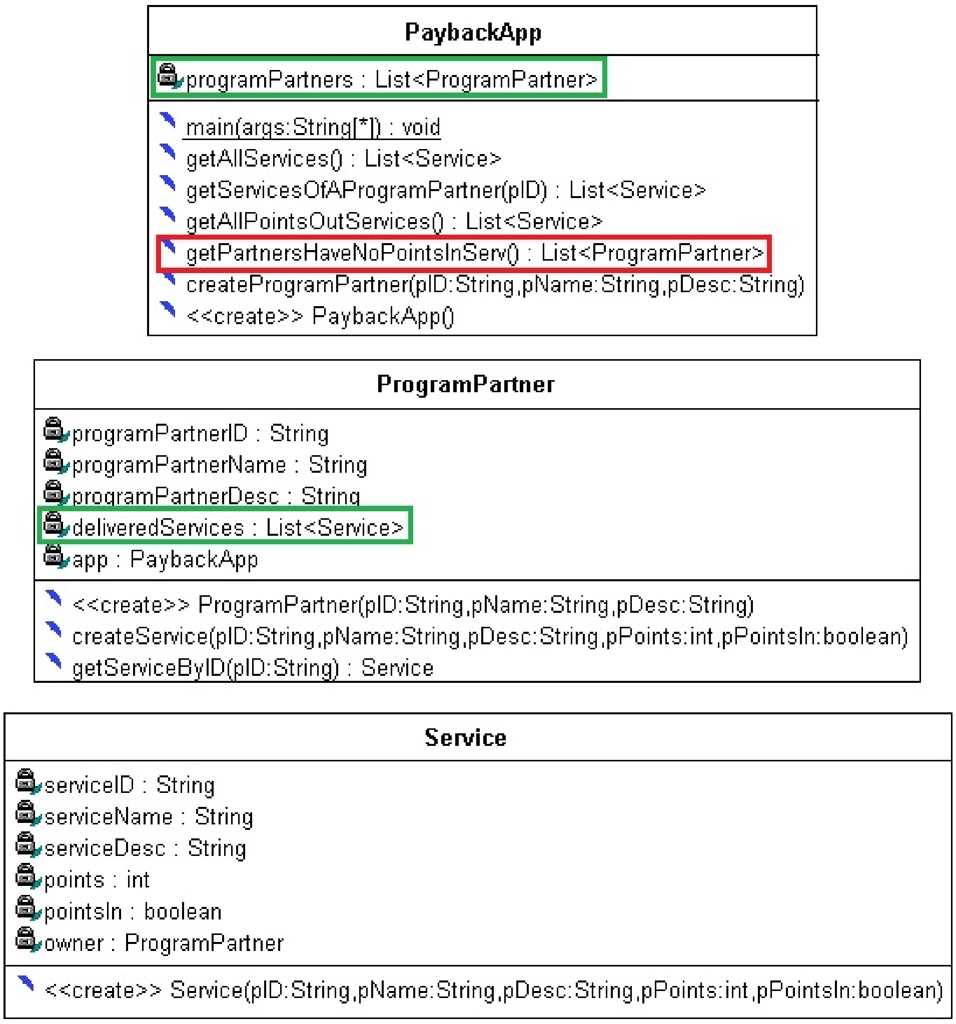

PaybackApp
programPartners : List<ProgramPartner>
main(args:String[*]) : void
getAllServices() : List<Service>
getServicesOfAProgramPartner(pID) : List<Service>
getAllPointsOutServices() : List<Service>
getPartnersHaveNoPointsInServ() : List<ProgramPartner>
createProgramPartner(pID:String,pName:String,pDesc:String)
<<create>> PaybackApp()
ProgramPartner
programPartnerID : String
programPartnerName : String
programPartnerDesc : String
deliveredServices : List<Service>
app : PaybackApp
<<create>> ProgramPartner(pID:String,pName:String,pDesc:String)
createService(pID:String,pName:String,pDesc:String,pPoints:int,pPointsIn:boolean)
getServiceByID(pID:String) : Service
Service
serviceID : String
serviceName : String
serviceDesc : String
points : int
pointsIn : boolean
owner : ProgramPartner
<<create>> Service(pID:String,pName:String,pDesc:String,pPoints:int,pPointsIn:boolean)

Fig. 4.18.: Target model: the corresponding target model on JSE platform of the design model in Figure 4.16

- the counterparts for each design model element have been generated in target model,
- the associations have been dissolved and represented as normal *attributes* in target model, which are denoted with the same green-colored rectangles,
- the counterparts for the typed-elements in design model are defined with types in JSE target model or target platform model,
- the access-modifiers are fixed and their getters and setters are also generated, which are not shown in Figure 4.18 to make model clean,
- the counterpart of the start-up operation enhanced by *«Main»* stereotype in design model is generated with the required signature by JSE platform.

For behavioral mapping, the *JavaStandardXOCLMapper* emits Java codes for all the operations, whose behaviors are modeled in XOCL expressions. As a concrete example, the Java code corresponds to the XOCL expression in Listing 4.15 is shown in Listing 4.16. Compared to the XOCL expression, the size of generated Java code is several times larger. However, it is not the case for structural mapping on JSE platform. Because the design model in Figure 4.16 is not strongly decorated by DPM stereotypes addressed in Section 3.3.4, which give model elements in class diagram more information that lead to sophisticated structural enhancement as e.g., depicted in Algrithm 4.5.

```
1  ArrayList<ProgramPartner> selectResult_0= new ArrayList<ProgramPartner>();
2  Iterator<ProgramPartner> itr_0=this.programPartners.iterator();
3  while(itr_0.hasNext()){
4    ProgramPartner itrVar_0= itr_0.next();
5    boolean forAllResult_0= true;
6    Iterator<Service> itr_1= itrVar_0.getDeliveredServices().iterator();
7    while(itr_1.hasNext()){
8      Service itrVar_1= itr_1.next();
9      forAllResult_0= forAllResult_0 && !itrVar_1.getPointsIn();
10   }
11   if(forAllResult_0)
12     selectResult_0.add(itrVar_0);
13 }
14 return selectResult_0;
```

Listing 4.16: Target model: Java code emitted by the JavaStandardXOCLMapper for the XOCL expression in Listing 4.15

Putting the emitted Java code and the abstract syntax tree in Figure 4.17 together, it is straightforward to understand that the *JavaStandardXOCLMapper* (as well as other implementation language specific XOCL mappers) is a *tree walker*, which traverses all the nodes in the underlying abstract syntax tree to generate the final Java code. Such a tree walker can be implemented either in classical recursive method based manner or by using *visitor pattern* [Gam+95] [Sch09], which is more object-oriented.

For each non-leaf AST-node, the Java code, which implements the semantics of that node, is generated by assembling the code pieces submitted by all its sub-nodes. For example, the code piece *this.programPartners* on the right hand side of the assignment in line 2 in Listing 4.16 is generated for the *FC2* node by assembling *this* submitted by traversing *PC1* node as the *caller* for the "." operator, and *programPartners* submitted by the *PC2* node as the *callee* for the "." operator.

The AST-nodes denoted in green background color represent the OCL predefined loop operation *select()* and *forAll()* respectively. As summarized in the second point at the beginning of Section 4.3.4, the mapping routines of original OCL predefined operations are implemented in the XOCL mapper knowing about a target language rather than specifying them in mapping configuration file. The mapping routines are template-based, which are inspired by the idea described in [WK03] and can be traced back to the early study in [Lia08]. The Java code templates installed in the *JavaStandardXOCLMapper* to map the original OCL *select()* and *forAll()* loop operations are shown in Listing 4.17 and Listing 4.18 respectively.

```
//Java code template for dmSrcColl -> select(boolExp)
tmSrcColl.rawTypeDef<tmSrcColl.contentType> selectResult_autoGenNum = new tmSrcColl.
    rawTypeDef<tmSrcColl.contentType>();
Iterator<tmSrcColl.contentType> itr_autoGenNum = tmSrcColl.resultLabel.iterator();
while(itr_autoGenNum.hasNext()){
   tmSrcColl.contentType itrVar_autoGenNum = itr_autoGenNum.next();
   evaluate boolExp;

   if(boolExp.resultLabel){
       selectResult_autoGenNum.add(itrVar_autoGenNum)
   }
}
```

Listing 4.17: Java code template for mapping OCL select() loop operation

```
1 //Java code template for dmSrcColl -> forAll(boolExp)
2 boolean forAllResult_autoGenNum = true;
3 Iterator<tmSrcColl.contentType> itr_autoGenNum = itr_autoGenNum.next();
4 while(itr_autoGenNum.hasNext()){
5    tmSrcColl.contentType itrVar_autoGenNum = itr_autoGenNum.next();
6    forAllResult_autoGenNum = forAllResult_autoGenNum && boolExp.resultLabel;
7 }
```

Listing 4.18: Java code template for mapping OCL forAll() loop operation

Comparing the emitted Java code in Listing 4.16 with the both template codes shown above, it is easy to find that the entire Java code in lines 1 to 13 corresponds to the *select()*-template with *forAll()*-template in lines 5 to 10 in Listing 4.16 embedded in it. This code represents a typical nested usage of OCL predefined operations, which can also be recognized in the abstract syntax tree in Figure 4.17, with *LP2* node the (indirect) right sub-node of *LP1* node.

An identifier denoted as *dmSrcColl* at Line 1 of the both templates represents the caller for the both loop operations. As required by XOCL, the *dmSrcColl* must have a collection type consisting of its raw type and content type. Concrete in this example, the *dmSrcColl*, on which *select()* is called, is the *programPartners* association-end, whose data type is *OrderedSet(ProgramPartner)*, whereas the *dmSrcColl*, on which *forAll()* is called, is the *deliveredServics* association-end, whose data type is *OrderedSet(Service)*. All this information have been filled in the corresponding AST-nodes by the *XOCLParser* as illustrated in Figure 4.17. At the time of *XOCL to Java mapping*, the *JavaStandardXOCLMapper* traverses the *PC2* node and consults the both mapping rule table and mapping table to find out that on JSE platform, the raw type of the *OrderedSet* collection is mapped to *ArrayList<T>* raw type, whereas the content type *ProgramPartner* is mapped to its counterpart generated by the JSE model mapper in the previous structural mapping. After traversing *PC2* node, the final Java code for the *FC2* node can be assembled. The *JavaStandardXOCLMapper* takes the callee data type, which in this case is *ArrayList<ProgramPartner>* submitted by traversing *PC2* node, as the final data type.

In the next step of mapping, all this information is passed to the *LP1* node, which represents the *select()* operation. The *ArrayList<T>* is used by the tree walker to replace all the *tmSrcColl.rawTypeDef* pseudo code in Listing 4.17, whereas *ProgramPartner* to replace *tmSrcColl.contentType*. Furthermore, the Java code *this.programPartners* emitted for *FC2* node is used to replace the *tmSrcColl.resultLabel* pseudo code in line 3 in Listing 4.17. The same process is also applied to the *LP2* node of *forAll()* operation. After that, the final code can be assembled for the *LP1* node by embedding the Java code generated for *LP2* node into the *select()*-template in lines 6 and 8 in Listing 4.17. It is worth noting that in line 11 in Listing 4.16 the *forAllResult_0*, which is a *part* of the code for *forAll()*-operation, is used by the *select()*-template. This phenomena indicates that both the *code segment* itself and the *result label*, which may participate in other code assembling, have to be kept and passed up and down by the tree walker.

In Listing 4.16, there are identifies named *itr_0* as well as *itrVar_0*, etc. These identifiers correspond to the ones with *autoGenNum*-suffix in the both templates. These identifiers are generated and maintained by the XOCL mapper automatically with the promise to avoid name conflict in the case of nested and concatenated invocations of OCL predefined operations.

4.4. Code Generation

Up to this point, a platform independent design model has been mapped onto a concrete target platform. The transformed internal model is saved as the target model, which contains the complete structure of the application in the form of the UML metaclass instances of packages, classifiers as well as their features and behaviors in the form of target language code saved in the opaque behaviors attached to the corresponding operations. Hence, the code generation from target model means traversal of all the model elements in target model for assembling code piece by piece. Depending on the peculiarities of different target languages, the *code generator* components targeted to them may function with slight differences. As shown in Figure 4.19, the *JavaStandardCodeGenerator* is used to illustrate the general architecture and to expose the common functionality of a code generator in MOCCA.

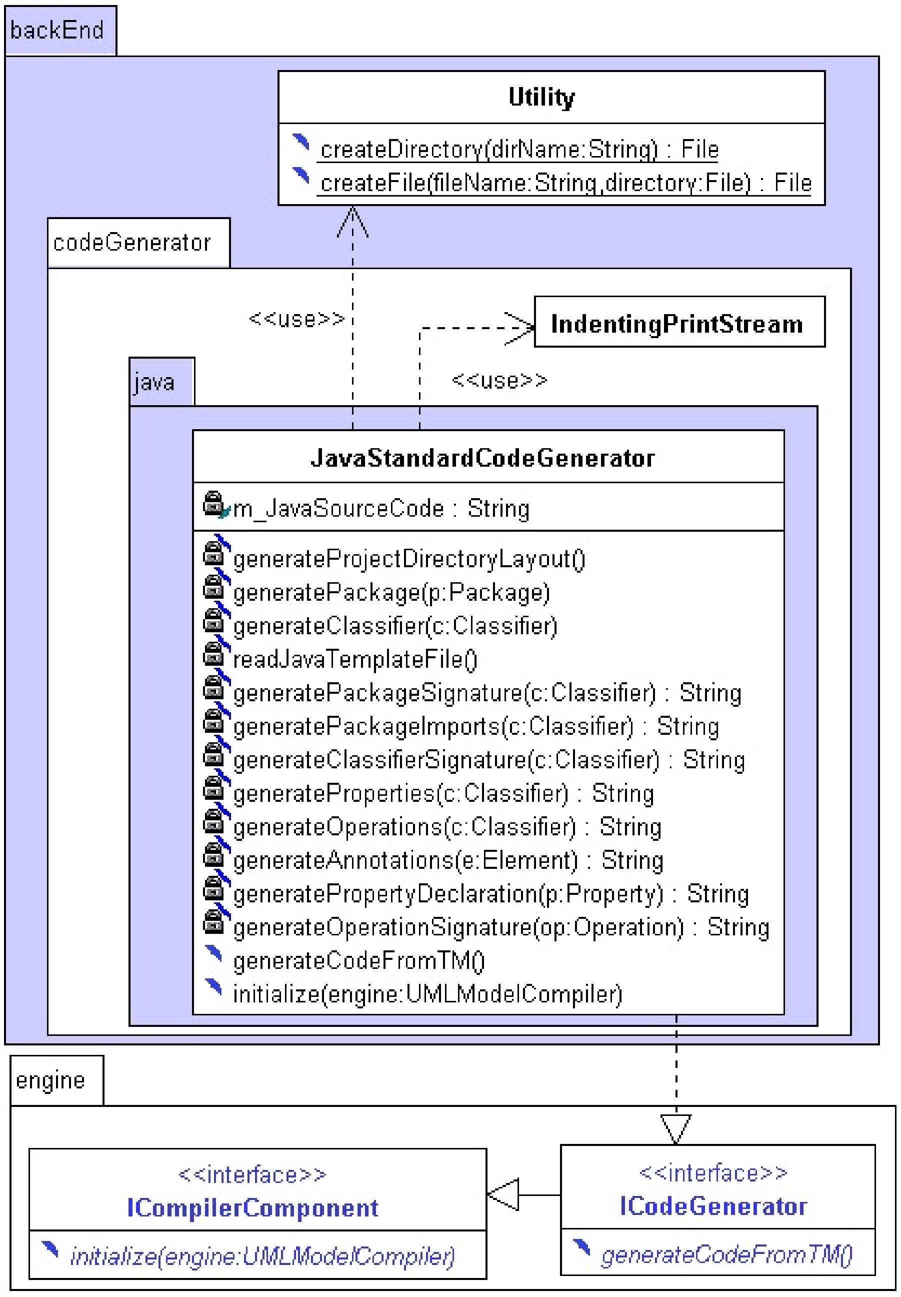

backEnd
Utility
createDirectory(dirName:String) : File
createFile(fileName:String,directory:File) : File
codeGenerator
<<use>>
IndentingPrintStream
<<use>>
java
JavaStandardCodeGenerator
m_JavaSourceCode : String
generateProjectDirectoryLayout()
generatePackage(p:Package)
generateClassifier(c:Classifier)
readJavaTemplateFile()
generatePackageSignature(c:Classifier) : String
generatePackageImports(c:Classifier) : String
generateClassifierSignature(c:Classifier) : String
generateProperties(c:Classifier) : String
generateOperations(c:Classifier) : String
generateAnnotations(e:Element) : String
generatePropertyDeclaration(p:Property) : String
generateOperationSignature(op:Operation) : String
generateCodeFromTM()
initialize(engine:UMLModelCompiler)
engine
<<interface>>
ICompilerComponent
initialize(engine:UMLModelCompiler)
<<interface>>
ICodeGenerator
generateCodeFromTM()

Fig. 4.19.: Architecture of the code generator component for the target language Java

The sole requirement for a code generator class is to implement the *ICodeGenerator* interface, which defines the *generateCodeFromTM()* operation that is the start point for the code generation process. First thing to do is to create the directory structure for the output. Depending on concrete target platform, this structure may vary. For Java based projects, the *SourcePackage* for saving Java source codes, the *Resources* for storing any resources used by the current project as well as the *Configuration* for maintaining deployment related artifacts are generated by the *generateProjectDirectoryLayout()* operation in *JavaStandardCodeGenerator* class. This operation retrieves the path of the project root directory from the *config.proj* file and calls the static helper operation *createDirectory()* provided in the *Utility* class to make directory physically.

As the next step, the *generatePackage()* operation is called with *target model* as input, which corresponds to the outermost package of the current project. Within this operation, all the *packageable elements* are iterated and for the packageable elements recognized as class or interface, the *generateClassifier()* operation is called with that element as input, whereas for the case of sub-package, the *generatePackage()* operation is called recursively for that package. Generating a package means creating another directory in the current directory, which is kept by the code generator class, whereas generating a classifier means creating a Java source file with *.java* as file extension. Again, the physical creation of a Java source file is achieved by calling the static *createFile()* utility operation.

Because the minimal code generation unit is a classifier, namely, a class or an interface. Hence, the final Java code is assembled for a classifier. The code is maintained in the *m_ JavaSourceCode* property of *JavaStandardCodeGenerator* and after flushing the code into the Java file for the current classifier, this property is emptied for the next classifier, whose code will be assembled. The code assembling process is the *string concatenation* that involves:

1. package signature, namely, the *package* statement in Java,
2. package importation, namely, the *import* statement in Java,
3. the optional annotations,
4. the classifier signature, namely, the *class* or *interface* keyword and other possible modifiers,
5. the property definitions,
6. the operation signatures as well as
7. the operation implementations for none-abstract operations.

The first six points listed above belong to the important Java code pieces that must be put together. Each of them will be generated by an dedicated operation in the *JavaStandardCodeGenerator* class with self-descriptive name as shown in Figure 4.19. It is worth noting that all these *generator operations* give back a string representing the Java code piece to be assembled in the *generateClassifier()* operation in correct order. The 7th point in the above list referring to the operation implementation, which is the most complicated task in the entire model transformation, has been done in model mapping. In code generation, the target language code representing an operation implementation is simply attached to the operation signature.

What needs to be clarified is that none of the current code generators is able to fulfill "pretty printing" for the generated code. However, the ultimate goal of MOCCA is complete application generation, no round-trip engineering is required to post-edit the generated code to complete an application. In fact, the generated code can be reviewed in any language-aware text editor conveniently, which usually supports formatting the code.

5. Experimental Results

5.1. The Simple Transcript Calculator

5.1.1. Problem Description

A desktop application should be developed, with which the students of the bachelor program *Angewandte Informatik* (BAI) (English: Applied Computer Science) of the Freiberg University of Technology and Mining can calculate the *current* achievements in their study. The BAI program consists of a wide spectrum of courses (also called *module*), which cover both theoretical foundations and diverse application fields in terms of the applied computer science. All these modules are documented in [BAI09a] and available online under the given address. To guide the students to establish their course plans in a reasonable way, correlated modules in terms of a knowledge unit are organized into *module groups*, which are documented in [BAI09b] and online available, too. The module groups are distinguished between compulsory and optional. To receive the degree from the BAI program, all the compulsory module groups and at least one of the optional module groups must be finished, and the required credits by that group must be collected. Credits are assigned to single module directly. If the examination of a module is passed, the corresponding credits are collected. In this way, the totally required credits can be collected for a module group that involves many modules.

Figure 5.1 shows the general structure described above. It is worth noting that all the names of modules and module groups are taken from [BAI09a] and [BAI09b] directly without translating them into English. For the purpose of test, the names can be considered just as the abstract symbols that distinguish modules. The first three module groups are compulsory, whereas the module groups with prefix *Anwendungsfach* are optional. To keep the representation simple and clear, neither all the modules belonging to a single module group nor all the optional module groups are shown in Figure 5.1. The **...** symbol is used to ignore them. As the optional module group *Anwendungsfach Geo* (Courses that are in terms of the application field geology) indicates that a module group can be nested. Exactly speaking for *Anwendungsfach Geo*, it consists of two sub-module-groups, namely the *Pflichtmodule Geo*, whose all involved modules must be finished, and the *Wahlpflichtmodule Geo*, whose required credits must be collected by finishing *enough* modules belonging to it. Hence, the module groups and their involved modules compose a tree structure as represented in Figure 5.1.

```
GINF.BA.Nr.13 Grundlagen_der_Informatik 9
DIGISYS1.BA.Nr.504 Digitale_Systeme_1 6
```

Listing 5.1: Properties of two modules saved in a plain text file

As a simple test example, it is supposed that all the elementary modules are saved in a plain text file with each line representing a module. The properties of a module are separated by the *space* symbol at a line. Listing 5.1 shows an example for two modules, each of which is composed of an *identifier*, a *name* as well as the *assigned credits.* To initialize the module groups, the application itself knows how to group the input modules into module groups by providing appropriate routine for that task.

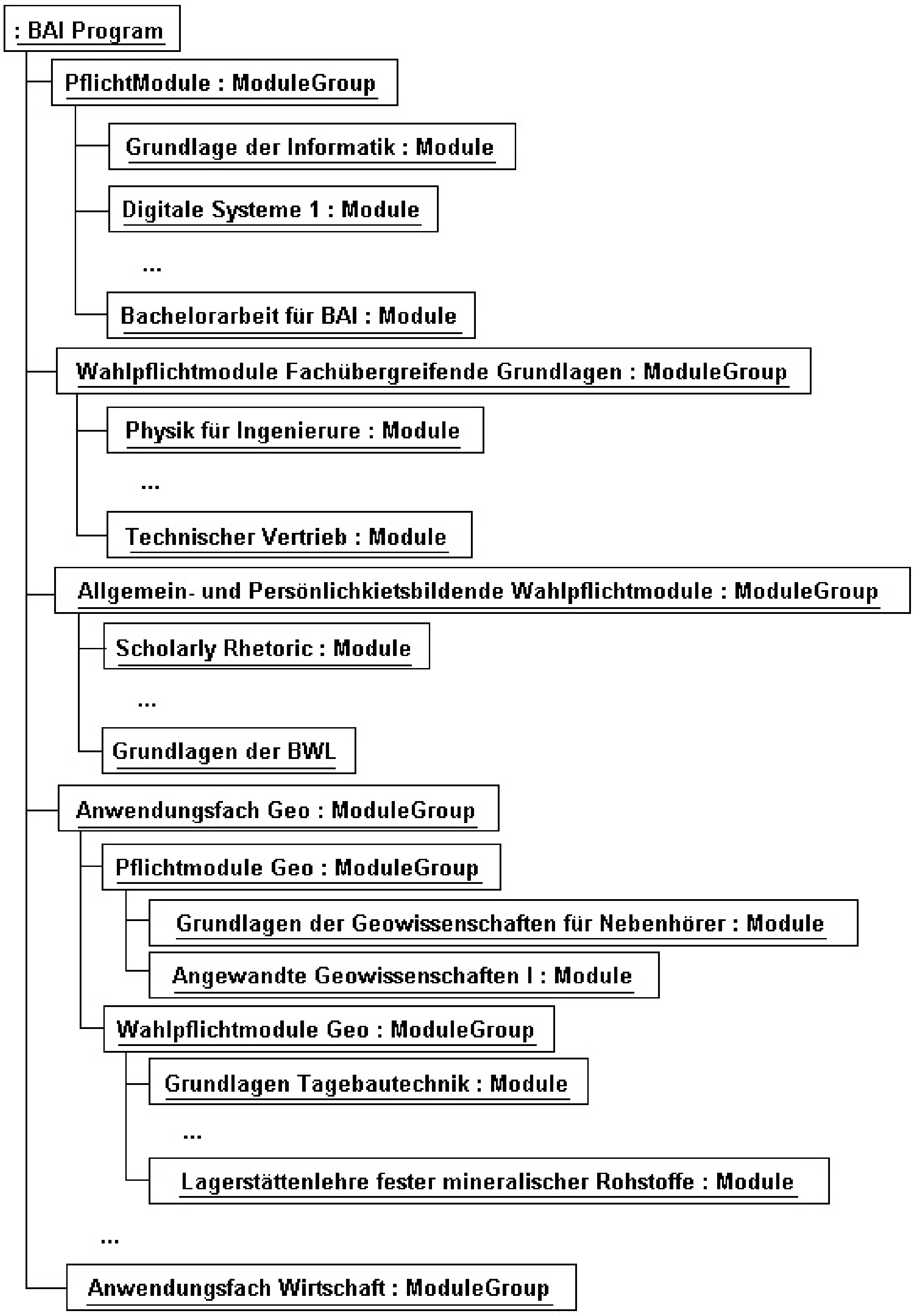
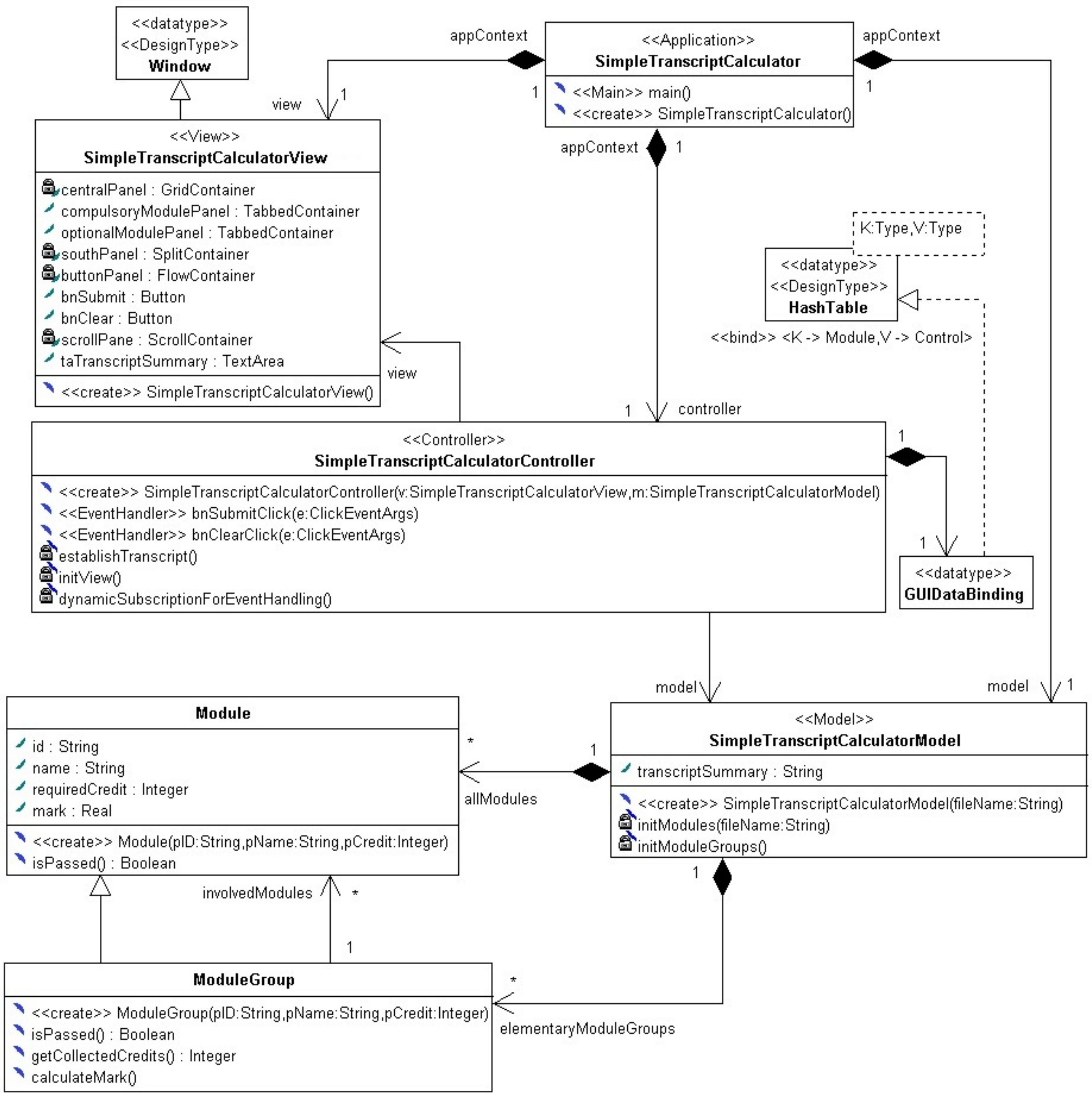

: BAI Program
PflichtModule : ModuleGroup
Grundlage der Informatik : Module
Digitale Systeme 1 : Module
...
Bachelorarbeit für BAI : Module
Wahlpflichtmodule Fachübergreifende Grundlagen : ModuleGroup
Physik für Ingenierure : Module
...
Technischer Vertrieb : Module
Allgemein- und Persönlichkietsbildende Wahlpflichtmodule : ModuleGroup
Scholarly Rhetoric : Module
...
Grundlagen der BWL
Anwendungsfach Geo : ModuleGroup
Pflichtmodule Geo : ModuleGroup
Grundlagen der Geowissenschaften für Nebenhörer : Module
Angewandte Geowissenschaften I : Module
Wahlpflichtmodule Geo : ModuleGroup
Grundlagen Tagebautechnik : Module
...
Lagerstättenlehre fester mineralischer Rohstoffe : Module
...
Anwendungsfach Wirtschaft : ModuleGroup

Fig. 5.1.: General structure of the module groups and their contained modules of the BAI program

5.1.2. The Design Model

With the background information given in the previous section in mind, the design issue of the simple transcript calculator can be addressed. To reflect the structure illustrated in Figure 5.1, the *Module* and *ModuleGroup* are modeled in the class diagram as shown in Figure 5.2. The properties of *Module* class ought to be intuitive. The *mark* is initialized with 0.0, whereas the others are read in from external file. The standard *constructor*s of the both classes take the input parameters to initialize their own state-properties. All the properties of *Module* are modeled with *public* access modifiers and moreover, the *id* of a *Module* is constrained by *read only*, so that only getter will be generated for *id*, whereas both getters and setters will be generated for the other properties.



<<datatype>>
<<DesignType>>
Window
appContext
<<Application>>
SimpleTranscriptCalculator
appContext
1
<<Main>> main()
<<create>> SimpleTranscriptCalculator()
1
view
1
<<View>>
SimpleTranscriptCalculatorView
centralPanel : GridContainer
compulsoryModulePanel : TabbedContainer
optionalModulePanel : TabbedContainer
southPanel : SplitContainer
buttonPanel : FlowContainer
bnSubmit : Button
bnClear : Button
scrollPane : ScrollContainer
taTranscriptSummary : TextArea
<<create>> SimpleTranscriptCalculatorView()
appContext
1
K:Type,V:Type
<<datatype>>
<<DesignType>>
HashTable
<<bind>> <K -> Module,V -> Control>
view
1
controller
<<Controller>>
SimpleTranscriptCalculatorController
<<create>> SimpleTranscriptCalculatorController(v:SimpleTranscriptCalculatorView,m:SimpleTranscriptCalculatorModel)
<<EventHandler>> bnSubmitClick(e:ClickEventArgs)
<<EventHandler>> bnClearClick(e:ClickEventArgs)
establishTranscript()
initView()
dynamicSubscriptionForEventHandling()
1
1
<<datatype>>
GUIDataBinding
model
model
1
Module
id : String
name : String
requiredCredit : Integer
mark : Real
<<create>> Module(pID:String,pName:String,pCredit:Integer)
isPassed() : Boolean
*
1
allModules
<<Model>>
SimpleTranscriptCalculatorModel
transcriptSummary : String
<<create>> SimpleTranscriptCalculatorModel(fileName:String)
initModules(fileName:String)
initModuleGroups()
1
involvedModules
*
1
ModuleGroup
<<create>> ModuleGroup(pID:String,pName:String,pCredit:Integer)
isPassed() : Boolean
getCollectedCredits() : Integer
calculateMark()
*
elementaryModuleGroups

Fig. 5.2.: Class diagram modeling the structure of the simple transcript calculator in design model

```
body:  self.mark<>0 and  self.mark<=4.0
```

Listing 5.2: XOCL body–expression specifying the query operation Module:: isPassed()

```
body:  self.getCollectedCredits() >=  self.requiredCredit
```

Listing 5.3: XOCL body–expression specifying the redefined query operation ModuleGroup:: isPassed()

```
body:  self.involvedModules->select(isPassed())->collect(requiredCredit)->sum()
```

Listing 5.4: XOCL body–expression specifying the query operation ModuleGroup:: getCollectedCredits()

Listing 5.2 specifies the *isPassed()* operation defined in *Module* class, which reflects the rule to check whether a module is finished, whereas Listing 5.3 implements the rule to check whether a module group is finished. To check finish of a module group, the credits collected until now must be known that corresponds to a query, which is specified in Listing 5.4.

```
1  begin
2     collCredits:Integer = self.getCollectedCredits();
3     if collCredits = 0
4     then begin
5       update self.mark = 5.0;
6     end
7     else begin
8       update self.mark = self.involvedModules
9                             ->select(isPassed())->collect(mark * requiredCredit)
10                                                   ->sum() / collCredits ;
11    end
12    endif
13 end
```

Listing 5.5: XOCL block–expression specifying the non–query operation ModuleGroup:: calculateMark()

$$Mark_{ModuleGroup} = \frac{Mark_{Module}[1] * Credit_{Module}[1] + ... + Mark_{Module}[n] * Credit_{Module}[n]}{Credit_{Module}[1] + ... + Credit_{Module}[n]} \tag{5.1}$$

The *calculateMark()* operation defined in *ModuleGroup* calculates the *current* mark for module group and saved the result in the *mark* property inherited from the base-class *Module*. Emphasizing "current", because for a none-finished module group, the current average mark will be calculated based on the *weighted average* method expressed in Equation 5.1. Because this operation is non-query, the XOCL *block*-expression must be used to specify its implementation logic as given in Listing 5.5. The *collect()* operation in line 9 returns all the weighted elements as a collection, on which the *sum()* operation is called to calculate the numerator in Equation 5.1.

To manage and maintain the application data at a central place, the *SimpleTranscriptCalculatorModel* class is introduced. Due to the applied *«Model»* stereotype, this class plays the role as model in the MVC design pattern. The both *composition associations* indicate that this class instantiates both the *modules* and the *module groups*. The single property *transcriptSummary* records the final report about the current achievement in the BAI study. The *private* operations *initModules()* and *initModuleGroups()* are called within the standard constructor to initialize the association-ends *allModules* and *elementaryModuleGroups* respectively. To specify the *initModules()* operation, the static XOCL library operation *readCollectionFromTextFile()* is called within an XOCL expression as illustrated in Listing 5.6.

```
begin
   update self.allModules = InputStream::readCollectionFromTextFile(fileName);
end
```

Listing 5.6: XOCL block–expression specifying the SimpleTranscriptCalculatorModel:: initModules()

After modules have been created, the *initModuleGroups()* operation classifies them into respective module groups. It is a *selection*-based logic depicted as in Listing 5.7. For a simple module group like *Pflichtmodule*, an instance is defined for that module group at first (line 2 in Listing 5.7), then its *involvedModules* property is filled with appropriate modules by selecting them from the *allModules* property initialized before (lines 3 to 8 in Listing 5.7). To keep listing clean, not all the selection criteria are given, the igored criteria are represented with **...** symbol. After filling modules into their owner module group, the fresh constructed module group is recorded into the *elementaryModuleGroup* property. For nested module group like *Anwendungsfach Geo*, its sub-module-groups are created firstly. Concretely speaking, the sub module group *Pflichtmdule Anwendungsfach Geo* is created in lines 11 to 18 in Listing 5.7 and the *Wahlpflichtmdule Anwendungsfach Geo* is created in lines 20 to 29 in Listing 5.7. After that, the owner module group can be created in lines 31 to 38 in Listing 5.7.

```
 1 begin
 2  BAI_PM: ModuleGroup = new ModuleGroup("BAI_PM", "Pflichtmodule", 132);
 3  update BAI_PM.involvedModules
 4         = self.allModules->select(id = "GINF.BA.Nr.13"          or
 5                                   id = "DIGISYS1.BA.Nr.504"     or
 6                                    ...
 7                                   id = "BAAINF.BA.Nr.983"
 8                                  );
 9  update self.elementaryModuleGroups->including(BAI_PM);
10
11  BAI_AF_GEO_PM: ModuleGroup = new ModuleGroup("BAI_AF_GEO_PM",
12                                               "Pflichtmodule Anwendungsfach Geo",
13                                               16);
14  update BAI_AF_GEO_PM.involvedModules
15     = self.allModules->select(   id = "GGEONEB.BA.Nr.124"   or
16                                  id = "ANWGEO1.BA.Nr.200"
17                              );
18  update self.elementaryModuleGroups->including( BAI_AF_GEO_PM);
19
20  BAI_AF_GEO_WPM: ModuleGroup = new ModuleGroup("BAI_AF_GEO_WPM",
21                                                "Wahlpflichtmodule Anwendungsfach Geo",
22                                                18);
23  update BAI_AF_GEO_WPM.involvedModules
24     = self.allModules->select( id = "MITGRUN.BA.Nr.722"   or
25                               id = "PDGLING.BA.Nr.516"   or
26                               ...
27                               id = "LGSTFMR.BA.Nr.628"
28                               );
29  update self.elementaryModuleGroups->including( BAI_AF_GEO_WPM);
30
31  BAI_AF_GEO: ModuleGroup= new ModuleGroup("BAI_AF_GEO",
32                                           "Anwendungsfach Geo",
33                                           34);
34  update BAI_AF_GEO.involvedModules
35     = self.elementaryModuleGroups->select( id = "BAI_AF_GEO_PM"   or
36                                            id = "BAI_AF_GEO_WPM"
37                                          );
38  update self.elementaryModuleGroups->including(BAI_AF_GEO);
39
40    ...
41 end
```

Listing 5.7: XOCL block–expression specifying the SimpleTranscriptCalculatorModel:: initModuleGroups()

Up to this point, all the classes concerning the application data structure have been addressed. As the next step, the GUI and the event handling mechanism of this application will be modeled. The *SimpleTranscriptCalculatorView* class represents the main window of this desktop application, whose properties are completely defined using the GUI elements provided in the GUI toolkit of the design platform model (Section 3.3.2). The properties, which have to be accessed from outside of the class, are defined using *public* modifiers, otherwise as private properties. Listing 5.8 shows the XOCL expressions declaring the GUI layout for the main window. A *GridContainer* is used to manage the both *TabbedContainer*s, which manage the compulsory module groups and optional module groups respectively. The XOCL expressions in lines 5 to 7 in Listing 5.8 model this structure. Furthermore, the both *Button*s and the *TextArea* for the output are put into a *FlowContainer* (lines 13 and 14) and a *ScrollContainer* (line 17) respectively. The both containers are positioned into an outer *SplitContainer* (lines 16 and 18), which lies at the bottom of the main window (line 21).

```
 1 begin
 2   update self.useDefaultCloseOperation = true;
 3   update self.setSize(1000, 800);
 4
 5   update self.centralPanel.setGrid(2, 1);
 6   update self.centralPanel.addChild(self.compulsoryModulePanel);
 7   update self.centralPanel.addChild(self.optionalModulePanel);
 8
 9   update self.southPanel.isHorizontal = false;
10
11   update self.bnSubmit.text = "Submit";
12   update self.bnClear.text  = "Clear";
13   update self.buttonPanel.addChild(self.bnSubmit);
14   update self.buttonPanel.addChild(self.bnClear);
15
16   update self.southPanel.firstPart = self.buttonPanel;
17   update self.scrollPane.viewPort  = self.taTranscriptSummary;
18   update self.southPanel.secondPart = self.scrollPane;
19
20   update self.workingArea.center = self.centralPanel;
21   update self.workingArea.bottom = self.southPanel;
22 end
```

Listing 5.8: XOCL expressions specifying the GUI Layout

```
begin
  update self.model.allModules
            ->iterate(itrVar_md |
                      begin
                        textField_md: TextField = self.GUIDataBinding->get(itrVar_md
                            );
                         sMark:String = textField_md.text;
                         if sMark <> ""
                         then begin
                           update itrVar_md.mark = sMark;
                         end
                         endif
                      end
                      );
  update self.establishTranscript();
end
```

Listing 5.9: XOCL expressions specifying the event handling operation SimpleTranscriptCalculatorController:: bnSubmitClick()

```
begin
  update self.model.transcriptSummary = "";
  update self.view.taTranscriptSummary.text ="";
  update self.model.allModules
              ->iterate(itrVar_md |
                        begin
                          update itrVar_md.mark = 0;
                          textField_md : TextField = self.GUIDataBinding->get(
                              itrVar_md);
                          update textField_md.text = "";
                        end
                       );
end
```

Listing 5.10: XOCL expressions specifying the event handling operation SimpleTranscriptCalculatorController:: bnClearClick()

```
begin
   event: self.view.bnSubmit.click ~  self.bnSubmitClick ;
   event: self.view.bnClear.click ~ self.bnClearClick ;
end
```

Listing 5.11: XOCL event–expressions connecting events to their handling operations

```
begin
 update self.model.elementaryModuleGroups
    ->iterate(itrVar_mg |
              begin
               update itrVar_mg.calculateMark();
               if itrVar_mg.isPassed() then begin
                  update self.model.transcriptSummary
                     = self.model.transcriptSummary.concat(itrVar_mg.name)
                               .concat(" ist bestanden. Gesamtnote: ")
                               .concat(itrVar_mg.mark)
                               .concat("\n");
               end
               else begin
                  update self.model.transcriptSummary
                     = self.model.transcriptSummary.concat(itrVar_mg.name)
                               .concat(" ist noch nicht bestanden. Es fehlt noch: ")
                               .concat(itrVar_mg.requiredCredit - itrVar_mg.
                                   getCollectedCredits())
                               .concat(" Leistungspunkte.\n")
                               .concat("Derzeitige Durchschnittsnote: ")
                               .concat(itrVar_mg.mark).concat("\n");
               end
               endif
               end
              );
  update self.view.taTranscriptSummary.text = self.model.transcriptSummary;
end
```

Listing 5.12: XOCL expressions specifying the non–query operation SimpleTranscriptCalculatorController:: establishTranscript()

As the name suggests and the *«Controller»* stereotype indicates, the *SimpleTranscriptCalculatorController* class is the controller according to the MVC pattern, which is equipped with required event handling methods and provides operation to connect events to their handling operations (the dynamic subscription process).

The event handling operations are enhanced by the *«EventHandler»* stereotypes in design model and specified in Listing 5.9 and Listing 5.10 respectively. In Listing 5.9, the extended XOCL *iterate* operation is called on *allModules* collection, whose semantics is identical to the generic *for-each*-loop coming with most modern OOPLs, which processes each collection-element retrieved by the given *iterator*-variable, here, the *itrVar_md*, by running the imperative code block within the *begin-end*-block. In design model (Figure 5.2), a *HashTable* named *GUI-DataBinding* with *Module* as its *Key* and *TextField* as its *value* is established and maintained. Hence, in each iteration step, the input text field corresponding to a module is queried and the input data kept by it is retrieved to update the *mark* property of a module. After all the marks are assigned to their modules, the *establishTranscript()* operation can be called to create the final report. The XOCL-expressions in Listing 5.10 are similar to those in Listing 5.9, but they reset the marks for all the modules and clear the text fields corresponding to them.

```
begin
  index_moduleGroup: Integer = 0;
  while index_moduleGroup < self.model.elementaryModuleGroups->size() begin
    itrVar_moduleGroup: ModuleGroup = self.model.elementaryModuleGroups->at(
        index_moduleGroup);
    numOfRows: Integer = itrVar_moduleGroup.involvedModules->size();
    panel_moduleGroup: GridContainer = new GridContainer();
    update panel_moduleGroup.setGrid(numOfRows, 2);

    if itrVar_moduleGroup.id = "BAI_PM" or  itrVar_moduleGroup.id = "BAI_WPM_FG" or
        itrVar_moduleGroup.id = "BAI_WPM_AP" then
    begin
       update self.view.compulsoryModulePanel.addTab( itrVar_moduleGroup.name,
           panel_moduleGroup);
    end
    endif

    if itrVar_moduleGroup.id = "BAI_AF_GEO" or  itrVar_moduleGroup.id = "BAI_AF_MAT"
        then
    begin
       update self.view.optionalModulePanel.addTab( itrVar_moduleGroup.name,
           panel_moduleGroup);
    end
    endif

    update self.GUIDataBinding->put(itrVar_moduleGroup, panel_moduleGroup);

    index_module: Integer = 0;
    while index_module < itrVar_moduleGroup.involvedModules->size() begin
      itrVar_module: Module = itrVar_moduleGroup.involvedModules->at( index_module);
      label_module: Label = new Label(itrVar_module.name);
      update panel_moduleGroup.addChild(label_module);

      ctrl: Control = self.GUIDataBinding->get(itrVar_module);
      if ctrl <> OclVoid then
      begin
         update panel_moduleGroup.addChild(ctrl);
      end
      else begin
          textField_module: TextField = new TextField();
          update textField_module.name = itrVar_module.name;
          update panel_moduleGroup.addChild(textField_module);
          update self.GUIDataBinding->put(itrVar_module, textField_module);
      end
      endif

      update index_module = index_module + 1;
    end
    endwhile
    update  index_moduleGroup=  index_moduleGroup +1;
  end
  endwhile

  update self.dynamicSubscriptionForEventHandling();
  update self.view.visible = true;
end
```

Listing 5.13: XOCL expressions initializing part of the GUI dynamically

Listing 5.11 models the behavior for the *dynamicSubscriptionForEventHandling()* operation. The both XOCL *event*-expressions connect the events to their handling methods. Listing 5.12 specifies the behavior for the *establishTranscript()* operation. Despite the length of Listing 5.12, the routine is a *string concatenation* within a for-each loop.

Listing 5.8 discussed before only defines the general GUI layout in that two *TabbedContainer* are used to manage the both compulsory and optional module groups with each *tab* in the respective tabbed container reserved for one module group. The concrete contents are not filled into the tab with the XOCL expressions in Listing 5.8. The GUI elements will be added to their owner tab dynamically based on the structure (nested or not) and the number of involved modules of a single module group. The *initView()* operation, whose behavior is specified as shown in Listing 5.13, takes over this responsibility. To understand this operation, it had better turn to the GUI window in Figure 5.5. For each module group (the outer *while*-expression in lines 3 to 47 in Listing 5.13), a $n \times 2$-*GridContainer* is created to manage its modules with n the number of involved modules (in lines 5 to 7 in Listing 5.13). Naturally, the fresh created *GridContainer* must be put into the tab belonging to the correct tabbed container (in lines 9 to 19 in Listing 5.13). For each module (the inner *while*-expression in lines 24 to 44), a *Label* (line 26) denotes its name and a *TextField* (line 35) takes the input as the mark of that module. Both GUI elements are organized into the grid container that is created before for the module group, which owns the current module. The text field for taking input for a module is indexed by that module in a property named *GUIDataBinding*, which is a hash table. This information is used to retrieve the mark from the text field and assign it to the correct module for further calculation. As shown in Figure 5.5, if the module group is nested, the grid containers representing the inner sub module groups take the place of the text fields, which appear in the case of non-nested module group.

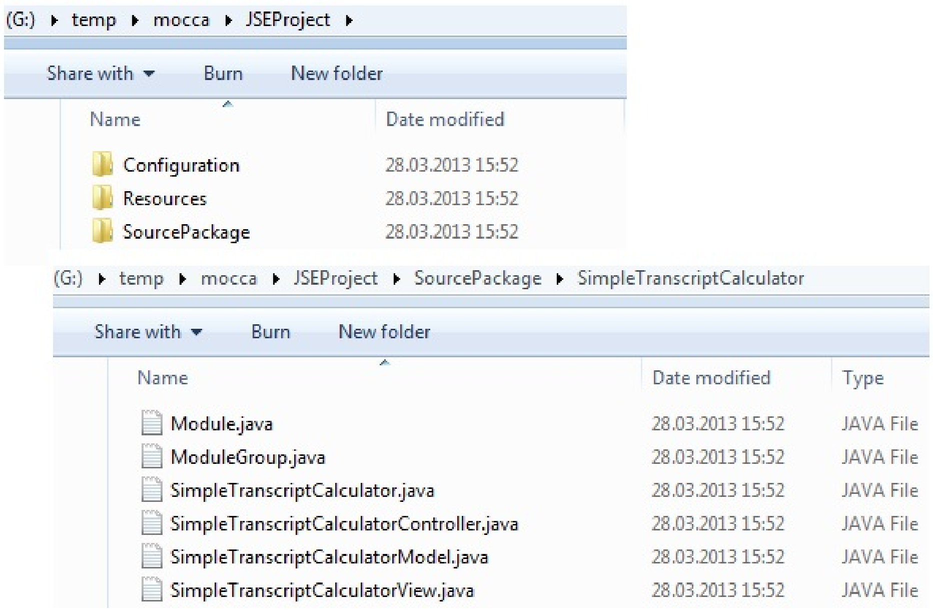
5.1.3. Transformation Result on the JSE Target Platform

The *simple transcript calculator* example application is mapped onto Java Standard Edition target platform as a desktop application. Listing 5.14 shows some JSE-specific configuration options. All these options are self-descriptive. It is worth noting that the value of *compiler.jdkPath* option is used to generate a batch-file as deployment descriptor for JSE application.

```
#
#Configuration For JSE Platform
#
compiler.jseModelMapper= compiler.backEnd.modelMapper.java.jse.JSEStandardModelMapper
compiler.jseCodeGenerator = compiler.backEnd.codeGenerator.java.
    JavaStandardCodeGenerator
compiler.jseCodeOutputFolder = G:\\temp\\mocca
compiler.jdkPath = C:\\Program Files\\Java\\jdk1.7.0_10\\bin
```

Listing 5.14: MOCCA configuration specific to JSE mapping

Figure 5.3 shows the generated directory structure and Java source files. An additional directory called *SimpleTranscriptCalculator* and residing in the *SourcePackage* directory denotes the root Java package for the currrent project. As illustrated in Figure 5.4, the generated batch file contains command line instructions that create a directory for Java class files, call the *javac* compiler as well as calling *java* to start the generated application.



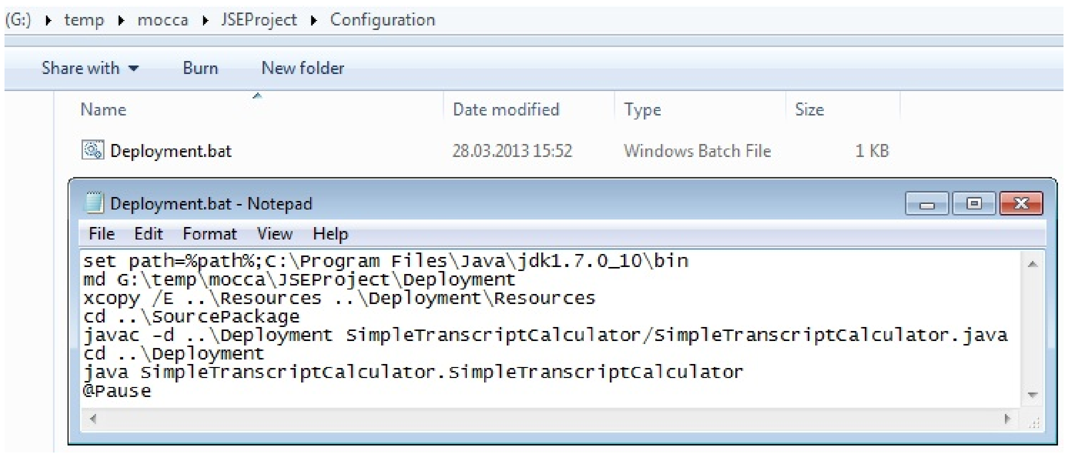
(G:) ▸ temp ▸ mocca ▸ JSEProject ▸

Name	Date modified
Configuration	28.03.2013 15:52
Resources	28.03.2013 15:52
SourcePackage	28.03.2013 15:52

(G:) ▸ temp ▸ mocca ▸ JSEProject ▸ SourcePackage ▸ SimpleTranscriptCalculator

Name	Date modified	Type
Module.java	28.03.2013 15:52	JAVA File
ModuleGroup.java	28.03.2013 15:52	JAVA File
SimpleTranscriptCalculator.java	28.03.2013 15:52	JAVA File
SimpleTranscriptCalculatorController.java	28.03.2013 15:52	JAVA File
SimpleTranscriptCalculatorModel.java	28.03.2013 15:52	JAVA File
SimpleTranscriptCalculatorView.java	28.03.2013 15:52	JAVA File

Fig. 5.3.: The generated directory structure of the simple transcript calculator example



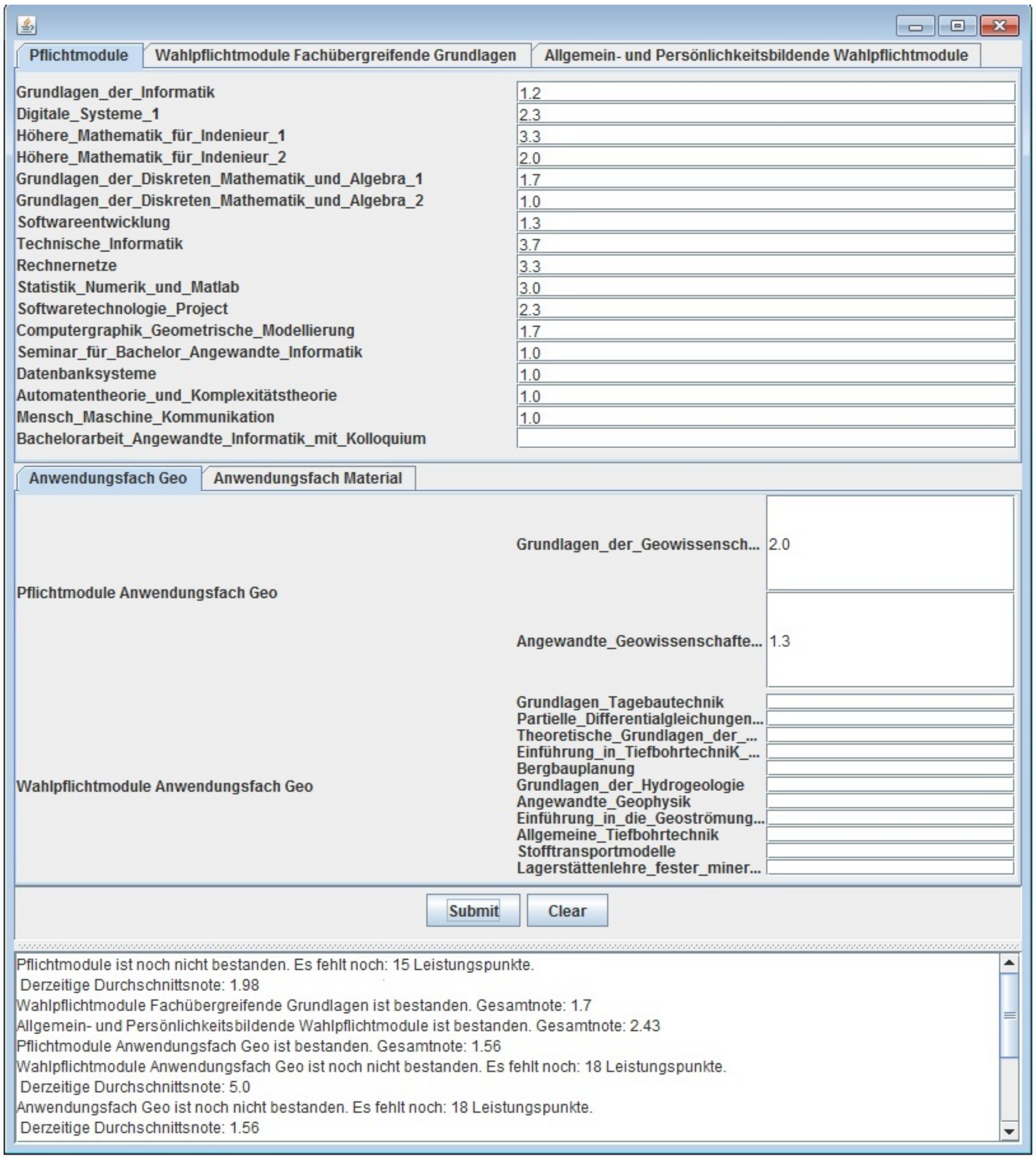
(G:) ▸ temp ▸ mocca ▸ JSEProject ▸ Configuration

Name	Date modified	Type	Size
Deployment.bat	28.03.2013 15:52	Windows Batch File	1 KB

```
set path=%path%;C:\Program Files\Java\jdk1.7.0_10\bin
md G:\temp\mocca\JSEProject\Deployment
xcopy /E ..\Resources ..\Deployment\Resources
cd ..\SourcePackage
javac -d ..\Deployment SimpleTranscriptCalculator/SimpleTranscriptCalculator.java
cd ..\Deployment
java SimpleTranscriptCalculator.SimpleTranscriptCalculator
@Pause
```

Fig. 5.4.: The generated batch file to compile and deploy the JSE application

Before the batch file is clicked to start the application, the text file recording the module information must be saved in the generated *Resources* directory, which will be copied together with all its contained files into a deployment directory saving Java binary files. The complete generated Java source code will not be listed in this work. Because the readers are strongly encouraged to run MOCCA on the given example model to generate this application by themselves. The same decision is also made for other test examples.


Pflichtmodule
Wahlpflichtmodule Fachübergreifende Grundlagen
Allgemein- und Persönlichkeitsbildende Wahlpflichtmodule

Grundlagen_der_Informatik	1.2
Digitale_Systeme_1	2.3
Höhere_Mathematik_für_Indenieur_1	3.3
Höhere_Mathematik_für_Indenieur_2	2.0
Grundlagen_der_Diskreten_Mathematik_und_Algebra_1	1.7
Grundlagen_der_Diskreten_Mathematik_und_Algebra_2	1.0
Softwareentwicklung	1.3
Technische_Informatik	3.7
Rechnernetze	3.3
Statistik_Numerik_und_Matlab	3.0
Softwaretechnologie_Project	2.3
Computergraphik_Geometrische_Modellierung	1.7
Seminar_für_Bachelor_Angewandte_Informatik	1.0
Datenbanksysteme	1.0
Automatentheorie_und_Komplexitätstheorie	1.0
Mensch_Maschine_Kommunikation	1.0
Bachelorarbeit_Angewandte_Informatik_mit_Kolloquium	

Pflichtmodule Anwendungsfach Geo	Grundlagen_der_Geowissensch...	2.0
	Angewandte_Geowissenschafte...	1.3
Wahlpflichtmodule Anwendungsfach Geo	Grundlagen_Tagebautechnik	
	Partielle_Differentialgleichungen...	
	Theoretische_Grundlagen_der_...	
	Einführung_in_TiefbohrtechniK_...	
	Bergbauplanung	
	Grundlagen_der_Hydrogeologie	
	Angewandte_Geophysik	
	Einführung_in_die_Geoströmung...	
	Allgemeine_Tiefbohrtechnik	
	Stofftransportmodelle	
	Lagerstättenlehre_fester_miner...	

```
Pflichtmodule ist noch nicht bestanden. Es fehlt noch: 15 Leistungspunkte.
 Derzeitige Durchschnittsnote: 1.98
Wahlpflichtmodule Fachübergreifende Grundlagen ist bestanden. Gesamtnote: 1.7
Allgemein- und Persönlichkeitsbildende Wahlpflichtmodule ist bestanden. Gesamtnote: 2.43
Pflichtmodule Anwendungsfach Geo ist bestanden. Gesamtnote: 1.56
Wahlpflichtmodule Anwendungsfach Geo ist noch nicht bestanden. Es fehlt noch: 18 Leistungspunkte.
 Derzeitige Durchschnittsnote: 5.0
Anwendungsfach Geo ist noch nicht bestanden. Es fehlt noch: 18 Leistungspunkte.
 Derzeitige Durchschnittsnote: 1.56
```

Fig. 5.5.: GUI of the generated simple transcript calculator desktop application

However, to show the benefit of modeling behaviors using XOCL, two transformed operations with their Java implementations are given. Listing 5.15 shows the corresponding Java implementation of the query operation *ModuleGroup::getCollectedCredits()*, whose behavior is modeled by an XOCL body-expression in Listing 5.4. The Java code exposes, to some extent, the working principle of the *JavaStandardXOCLMapper*. The XOCL expression in Listing 5.4 is a *concatenation* of three XOCL collection operations. As discussed in Section 4.3.5, the final Java code is assembled with the three pieces of Java code emitted by traversing the according AST-nodes. Hence, the Java code in lines 5 to 12 in Listing 5.15 corresponds the template code for *select()*-operation, the code in lines 13 to 17 together with line 4 represents the template for *collect()*-operation, and the code in lines 18 to 21 including line 3 embodies the usage of the *sum()*-template, respectively. Listing 5.16 shows the Java code generated from the XOCL event-expressions in Listing 5.11.

```
public int getCollectedCredits()
{
  int sumResult_0 = 0;
  ArrayList<Integer> collectResult_0 = new ArrayList<Integer>();
  ArrayList<Module> selectResult_0 = new ArrayList<Module>();
  Iterator<Module> itr_0 = this.getInvolvedModules().iterator();
  while(itr_0.hasNext()) {
     Module itrVar_0 = itr_0.next();
     if(itrVar_0.isPassed()){
         selectResult_0.add(itrVar_0);
     }
  }
  Iterator<Module> itr_1 = selectResult_0.iterator();
  while(itr_1.hasNext()){
     Module itrVar_1 = itr_1.next();
        collectResult_0.add(itrVar_1.getRequiredCredit());
  }
  Iterator<Integer> itr_2 = collectResult_0.iterator();
  while(itr_2.hasNext()){
     sumResult_0 += itr_2.next();
  }
  return sumResult_0;
}
```

Listing 5.15: The generated Java code corresponding to the XOCL body–expression in Listing 5.4

```
private void dynamicSubscriptionForEventHandling()
{
   this.view.getBnSubmit().addActionListener(
         new ActionListener(){
         @Override
         public void actionPerformed(ActionEvent e){
            bnSubmitClick(e);
   }});

   this.view.getBnClear().addActionListener(
         new ActionListener(){
         @Override
         public void actionPerformed(ActionEvent e){
            bnClearClick(e);
   }});
}
```

Listing 5.16: The generated Java code corresponding to the XOCL event–expressions in Listing 5.11

5.1.4. Evaluation

In this experiment, the design model in Figure 5.2 is mapped on the JSE target platform as a desktop application. As Figure 5.6 shows, the structural mapping is done in almost a one-to-one manner. There is one more data type, namely, the *GUIDataBinding*, created in the design model for concretizing the parameterized *HashTable<K, V>*. The *GUIDataBinding* is mapped as a property of the *controller* class due to the composition association between them. In Java implementation, more methods are generated as getters and setters due to the *public* properties in design model. Although from the quantitative perspective, the structure mapping targeted to JSE platform does not add many artifacts, the design model in Figure 5.2 presents a clear overview of the underlying structure, which is completely platform independent.

To evaluate the mapping result of behaviors, which are specified by XOCL expressions in design model, methods defined in two design classes are used. As shown in Figure 5.7, *SimpleTranscriptCalculatorController* contains non-query operations, whose behaviors can only be specified by the imperative enhancements of the XOCL. In this case, the XOCL enhancement

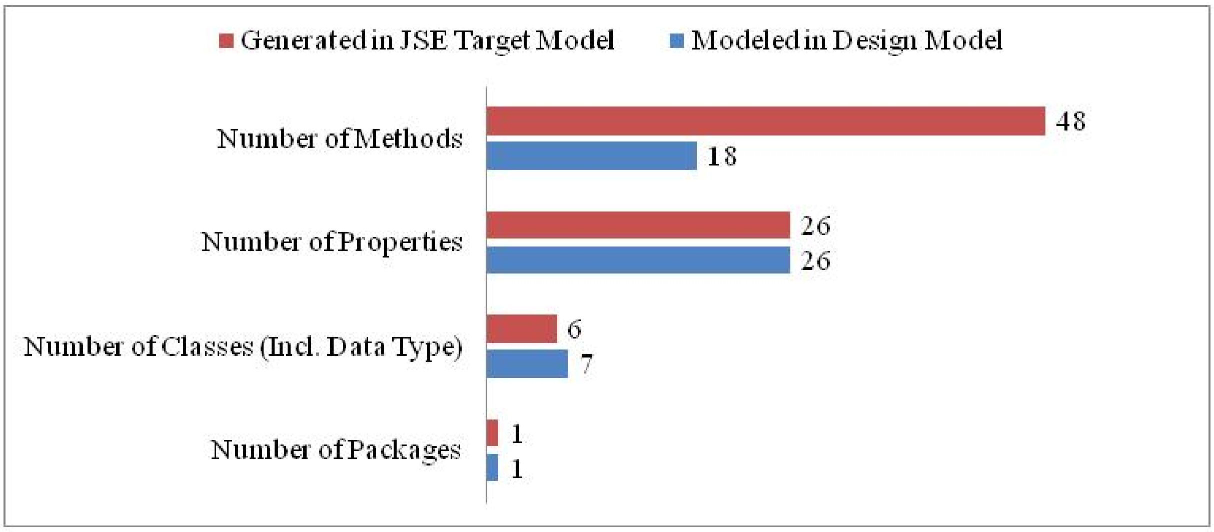


Fig. 5.6.: Evaluation of the structural model and its corresponding source code

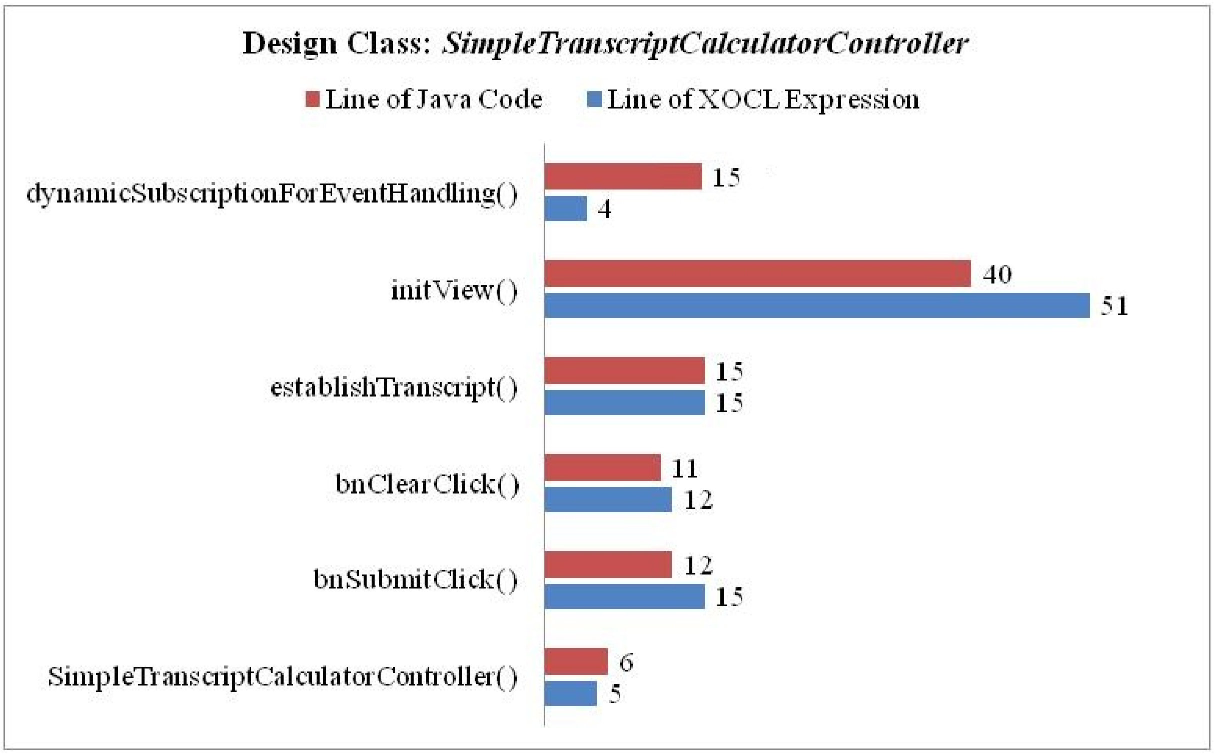


Fig. 5.7.: Evaluation of the behavioral specification and its corresponding source code for non-query operations

provides a way, in which non-query semantics can be specified with an OCL-like syntax. As illustrated here, in most situations, the size of generated Java code should be almost the same as the used XOCL expressions. As shown in Listing 5.12, whose XOCL expressions specify the behavior of the *establishTranscript()* operation, long XOCL expressions are usually split into several lines to improve the readability, that can lead to the circumstances with larger size of XOCL expression than the generated Java code. Because the current Java code generator takes into account little about readability of the generated Java code. As presented in Figure 5.7, if the XOCL expressions of *establishTranscript()* are formatted similar to the generated Java code,

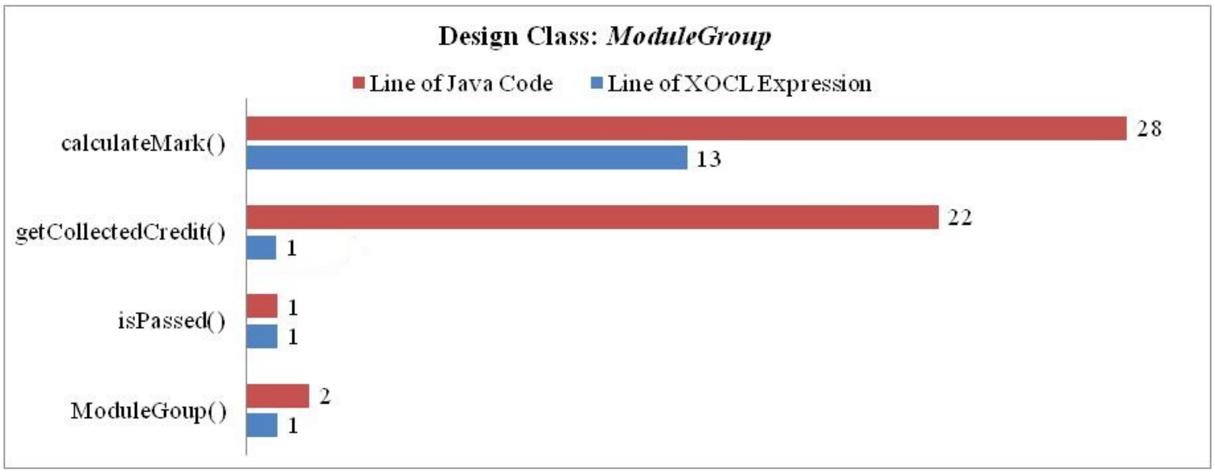


Fig. 5.8.: Evaluation of the behavioral specification and its corresponding source code for query and quasi query operations

both of them have the identical size. The *dynamicSubscriptionForEventsHandling()* method is specified using XOCL *event expressions*, which condense the information obviously.

As shown in Figure 5.8, the behaviors of the query operation like *getCollectedCredit()* and the quasi-query operation like *calculateMark()*, whose primary logic is specified using XOCL library operations, can be specified using XOCL expressions efficiently.

5.2. The Simple Payback System

5.2.1. Problem Description

Inspired by the "Royal and Loyal System" example from [WK03], a *Simple Payback System* should be developed as an enterprise application, which can handle concurrency management, persist data into relational database, be accessed by diverse front-end applications, say, web-applications or desktop applications distributed at any physical locations with Internet access.

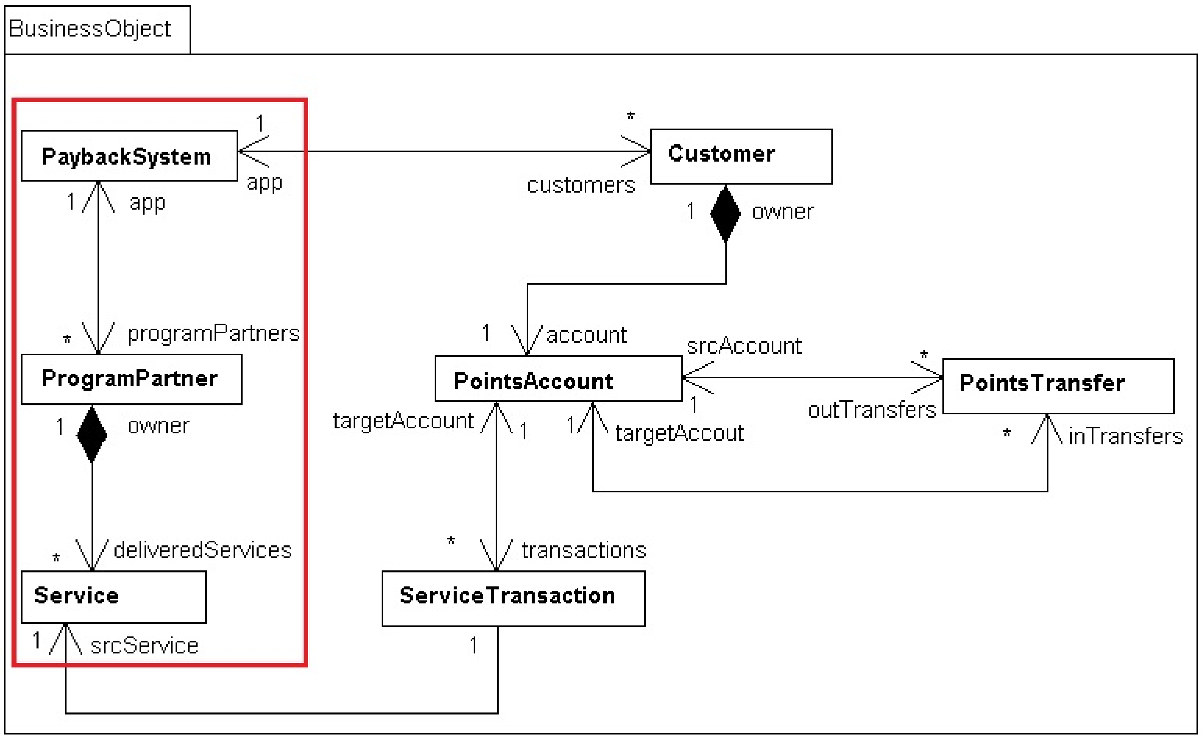


Fig. 5.9.: The initial data model of the simple payback system

This payback system is an information system that manages the registered program partners and customers as well as their transactions. Any company offering services to customers can register into the payback system as a *program partner*, whose identification information must be kept by the system. After registration, a program partner can launch, modify and cancel diverse *services*. From the perspective of customers there are services, with which they can obtain *bonus points* as well as services, with which they must spend bonus points. Any single person, who will enjoy the payback service, can register into the payback system as a *customer* by submitting necessary identification information. After registration, an *account* to record the actual bonus points collected by that customer is created. Arbitrary services can be taken on an account in the form of *transactions*, which cause the increasing or decreasing of the bonus points based on the kind of services taken. Additionally, the bonus points, which can be considered as the virtual currency within the payback system, can be *transferred* between two payback accounts.

Based on the description above, the initial data model representing the essential structure of the payback system can be created as given in Figure 5.9. Naturally, this model is a PIM according to MDA but not yet a full-fledged design model according to MOCCA. With several refinements, important properties, operations will be added, certain model elements will be stereotyped with special semantics, etc. In order to make the resulted design model more readable and concentrate on exploring essential ideas, only the three classes highlighted by a rectangle with red frame color will be refined and shown, other classes just follow the same principle.

5.2.2. The Design Model

The three selected design classes have been refined and completed with necessary information as shown in Figure 5.10. It is clear that the refined design model is based on the three layer architecture and the current example concentrates on the business object layer and persistence layer, which will be deployed on the same application server. Thanks to the layered methodology, it is up to the GUI- or web-designer to model and conceive a best suited presentation to access the application logic.

```
body: self.programPartners
```

Listing 5.17: Behavior of the query operation PaybackSystem:: getAllProgramPartners()

```
body: self.programPartners->any( programPartnerID = pID )
```

Listing 5.18: Behavior of the query operation PaybackSystem:: getProgramPartnerByID()

```
body: self.getAllProgramPartners()
                    ->select( deliveredServices
                                        ->forAll(not pointsIn) )
```

Listing 5.19: Behavior of the query operation PaybackSystem:: getPartnersHaveNoPointsInServ()

```
body: self.programPartners.deliveredServices
```

Listing 5.20: Behavior of the query operation PaybackSystem:: getAllServices()

```
body: self.getProgramPartnerByID(pID).deliveredServices
```

Listing 5.21: Behavior of the query operation PaybackSystem:: getServicesOfAProgramPartner()

```
body: self.getAllServices()->select( not pointsIn)
```

Listing 5.22: Behavior of the query operation PaybackSystem:: getAllPointsOutServices()

Stereotyped by the *«Application»*, the *PaybackSystem* is the kernel of the entire system, which initializes and coordinates all the other components of the system. In the entire life cycle of the system, there is only one instance of this class. In a distributed system like this, operations can be classified in two groups: operations called only locally and operations called by different actors from remote clients. Other special operations include the start-up operation for the entire system, the constructors as well as the required setters and getters for the properties of the classes. In the class *PaybackSystem*, all the operations stereotyped by *«CommonOperation»* are bound to the *«CommonRole»*. As explained in Section 3.3.4, the common role indicates anonymous system users without registration, and their callable operations are usually general purpose query operations, which can be specified by XOCL expressions very efficiently. Listing 5.17 to Listing 5.22 show the behaviors of all these common operations. The common operations are *remote* callable, but due to their non-writable characteristics, they are able to be invoked concurrently.

In contrast to the common operations belonging to common role, the *«BusinessOperation»* in *PaybackSystem*-class can be called only by a special system actor declared by *«AdminRole»*. Listing 5.23 and Listing 5.24 show the behaviors of the both business operations defined for the admin role. In fact, all these operations (common and business operations) build the CRUD (Create, Read, Update and Delete) operations that are ubiquitous for information systems. For a real system, all these operations have to be provided, whereas for a simple demonstration example like this, the delete operation is omitted.

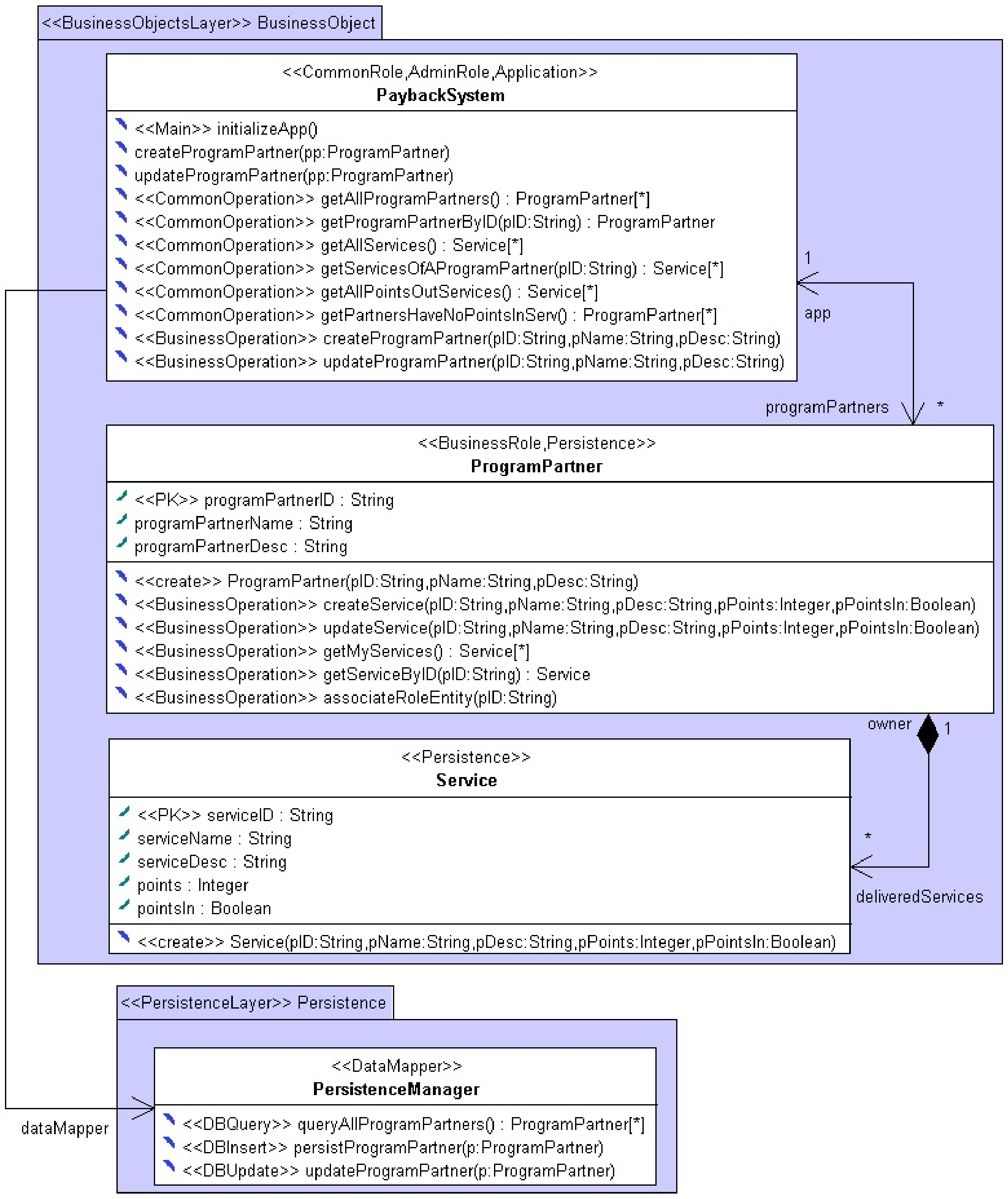

<<BusinessObjectsLayer>> BusinessObject
<<CommonRole,AdminRole,Application>>
PaybackSystem
<<Main>> initializeApp()
createProgramPartner(pp:ProgramPartner)
updateProgramPartner(pp:ProgramPartner)
<<CommonOperation>> getAllProgramPartners() : ProgramPartner[*]
<<CommonOperation>> getProgramPartnerByID(pID:String) : ProgramPartner
<<CommonOperation>> getAllServices() : Service[*]
<<CommonOperation>> getServicesOfAProgramPartner(pID:String) : Service[*]
<<CommonOperation>> getAllPointsOutServices() : Service[*]
<<CommonOperation>> getPartnersHaveNoPointsInServ() : ProgramPartner[*]
<<BusinessOperation>> createProgramPartner(pID:String,pName:String,pDesc:String)
<<BusinessOperation>> updateProgramPartner(pID:String,pName:String,pDesc:String)
1
app
programPartners
*
<<BusinessRole,Persistence>>
ProgramPartner
<<PK>> programPartnerID : String
programPartnerName : String
programPartnerDesc : String
<<create>> ProgramPartner(pID:String,pName:String,pDesc:String)
<<BusinessOperation>> createService(pID:String,pName:String,pDesc:String,pPoints:Integer,pPointsIn:Boolean)
<<BusinessOperation>> updateService(pID:String,pName:String,pDesc:String,pPoints:Integer,pPointsIn:Boolean)
<<BusinessOperation>> getMyServices() : Service[*]
<<BusinessOperation>> getServiceByID(pID:String) : Service
<<BusinessOperation>> associateRoleEntity(pID:String)
owner
1
<<Persistence>>
Service
<<PK>> serviceID : String
serviceName : String
serviceDesc : String
points : Integer
pointsIn : Boolean
<<create>> Service(pID:String,pName:String,pDesc:String,pPoints:Integer,pPointsIn:Boolean)
*
deliveredServices
<<PersistenceLayer>> Persistence
<<DataMapper>>
PersistenceManager
<<DBQuery>> queryAllProgramPartners() : ProgramPartner[*]
<<DBInsert>> persistProgramPartner(p:ProgramPartner)
<<DBUpdate>> updateProgramPartner(p:ProgramPartner)
dataMapper

Fig. 5.10.: Structural model of the simple payback system in its design model

```
begin
   p:ProgramPartner = new ProgramPartner(pID, pName, pDesc);
   update self.createProgramPartner(p);
end
```

Listing 5.23: Behavior of the non–query operation PaybackSystem:: createProgramPartner()

```
1 begin
2    p:ProgramPartner = self.getProgramPartnerByID(pID);
3    update p.programPartnerName = pName;
4    update p.programPartnerDesc = pDesc;
5    update self.updateProgramPartner(p);
6 end
```

Listing 5.24: Behavior of the non–query operation PaybackSystem:: updateProgramPartner()

In both remote callable business operations shown above, the actual creation (line 3 in Listing 5.23) and update (line 5 in Listing 5.24) of a program partner are delegated to the respective local operations, which take the object reference as parameter rather than elementary data as in remote operations. Such local operations belong to the application class rather than any specific *role* and can be considered as some kind of *auxiliary* or *utility* operations. Listing 5.25 specifies the actual creation operation that firstly adds the fresh created program partner into the system cache and then persists it into the underlying database via the *persistence manager*, which is modeled in the *persistence layer* by using *«ModelMapper»* stereotype and only accessable by the business object layer via the property *dataMapper* as illustrated in Figure 5.10. The similar behavior is specified for the actual update operation shown in Listing 5.26 with an additional check that guarantees the corresponding object update in the system cache.

```
begin
  update   self.programPartners->including(pp);
  update   self.dataMapper.persistProgramPartner(pp);
end
```

Listing 5.25: Behavior of the non–query local operation PaybackSystem:: createProgramPartner()

```
begin
  if self.programPartners->excludes(pp)
  then
     begin
       p:ProgramPartner = self.programPartners
                                        ->any(programPartnerID = pp.programPartnerID);
       update p.programPartnerName = pp.programPartnerName;
       update p.programPartnerDesc = pp.programPartnerDesc;
       update p.deliveredServices = pp.deliveredServices;
     end
  endif

  update self.dataMapper.updateProgramPartner(pp);
end
```

Listing 5.26: Behavior of the non–query local operation PaybackSystem:: updateProgramPartner()

```
begin
  update self.programPartners = self.dataMapper.queryAllProgramPartners();
end
```

Listing 5.27: Behavior of the system start–up operation PaybackSystem:: initializeApp()

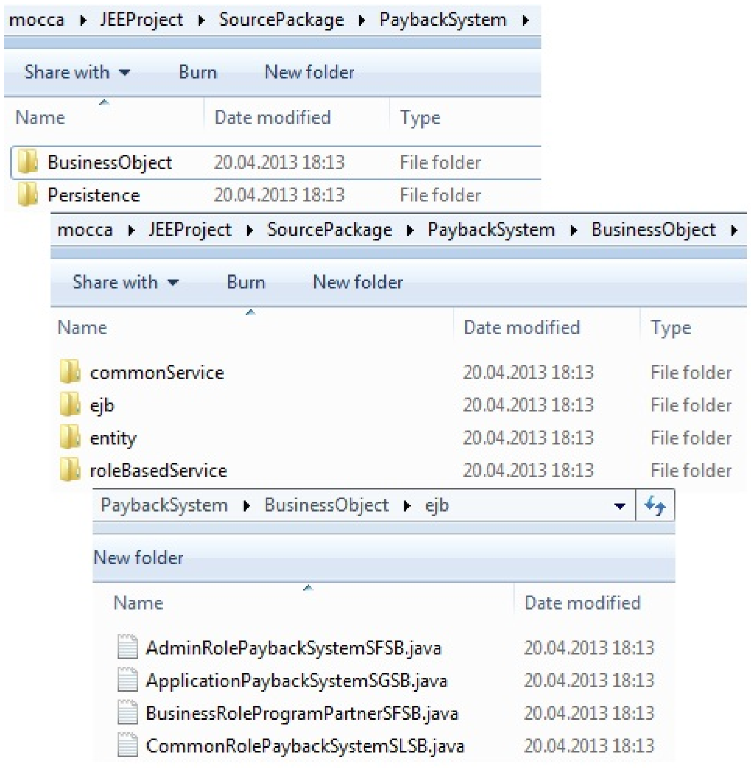
The three data base related operations defined in the *PersistenceManager* class are self-descriptive. They are enhanced by the corresponding stereotypes, which have been already addressed in Section 3.3.4. If the default semantics of these stereotypes are reserved, none additional behavioral modeling are necessary for these operations.

The entrance operation for the entire system is the *initializeApp()* in *PaybackSystem*, which is enhanced by the *«Main»* stereotype. Listing 5.27 shows its behavior, which is nothing more than the setup of the system cache of important data entities. For the current example model, only the cache for program partners is required.

The class *ProgramPartner* represents not only the data entity kept by the system as master data to identify a program partner, but also a system actor, who can invoke some business operations to do his job. Recall from Section 3.3.4, the *«Persistence»* stereotype is used to mark a design class, whose properties must be stored in external medium. The property marked by the *«PK»* stereotype indicates the identifier for a data entity. The *«BusinessRole»* stereotype enhances a design class with the semantics of a potential system actor, whose available interactions with the system are defined by the operations enhanced by the *«BusinessOperation»* stereotype. The most business operations of the *ProgramPartner* class are self-descriptive and moreover, their behaviors are similar to the ones in *PaybackSystem* very much. Hence, a complete listing of all these operations is not that valuable. Furthermore, the *«Persistence»*-class *Service* is simple, a detailed explanation is considered as unnecessary.

5.2.3. Transformation Result on the JEE Target Platform

The *simple payback system* design model can be mapped onto Java Enterprise Edition target platform as a distributed enterprise application. Figure 5.11 shows the generated package structure and the EJB source files.


mocca ▸ JEEProject ▸ SourcePackage ▸ PaybackSystem ▸
Share with ▾
Burn
New folder

Name	Date modified	Type
BusinessObject	20.04.2013 18:13	File folder
Persistence	20.04.2013 18:13	File folder

mocca ▸ JEEProject ▸ SourcePackage ▸ PaybackSystem ▸ BusinessObject ▸
Share with ▾
Burn
New folder

Name	Date modified	Type
commonService	20.04.2013 18:13	File folder
ejb	20.04.2013 18:13	File folder
entity	20.04.2013 18:13	File folder
roleBasedService	20.04.2013 18:13	File folder

PaybackSystem ▸ BusinessObject ▸ ejb
New folder

Name	Date modified
AdminRolePaybackSystemSFSB.java	20.04.2013 18:13
ApplicationPaybackSystemSGSB.java	20.04.2013 18:13
BusinessRoleProgramPartnerSFSB.java	20.04.2013 18:13
CommonRolePaybackSystemSLSB.java	20.04.2013 18:13

Fig. 5.11.: The generated package structure and the EJB source files on JEE platform

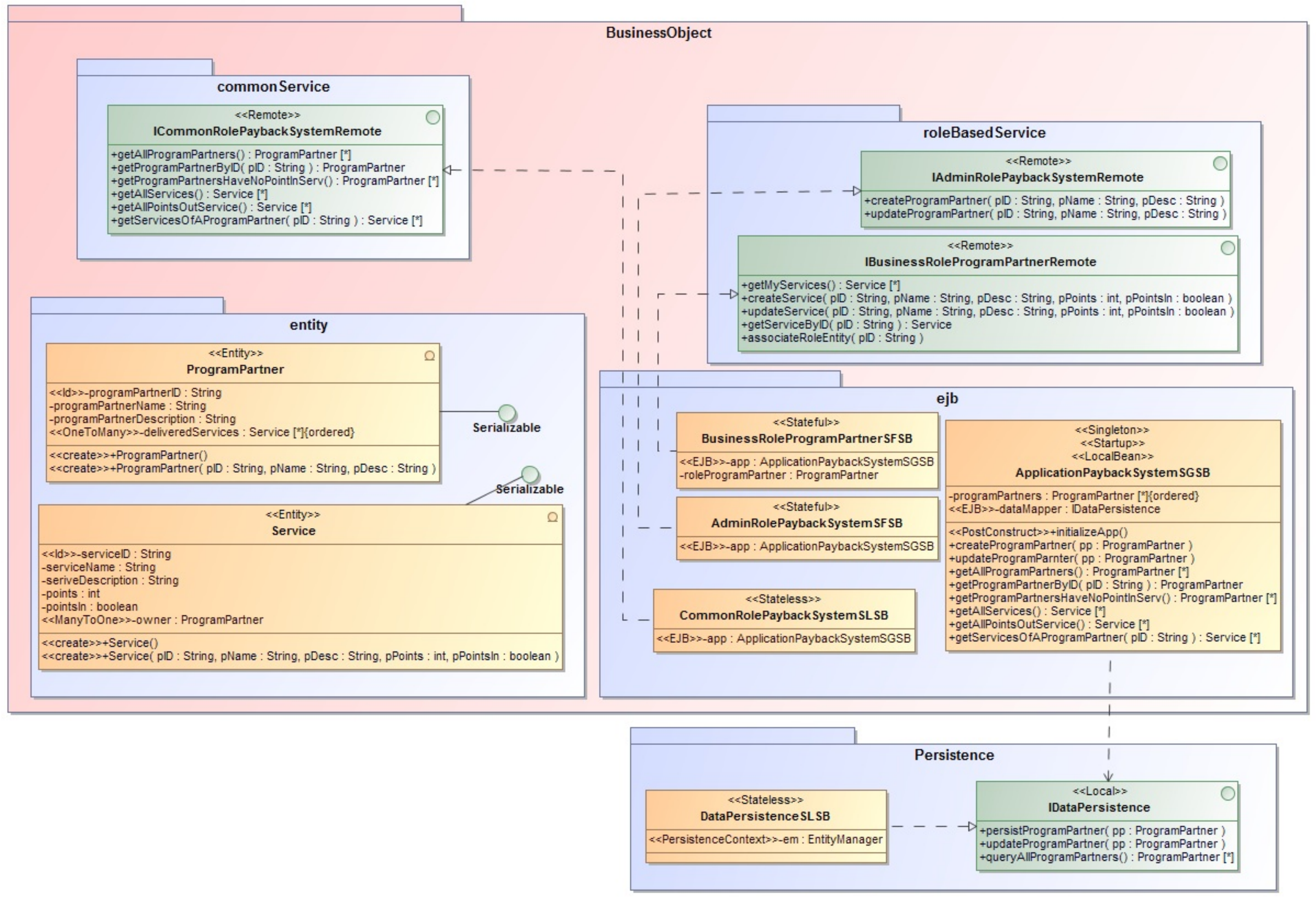

BusinessObject
commonService
<<Remote>>
ICommonRolePaybackSystemRemote
+getAllProgramPartners() : ProgramPartner [*]
+getProgramPartnerByID(pID : String) : ProgramPartner
+getProgramPartnersHaveNoPointInServ() : ProgramPartner [*]
+getAllServices() : Service [*]
+getAllPointsOutService() : Service [*]
+getServicesOfAProgramPartner(pID : String) : Service [*]
roleBasedService
<<Remote>>
IAdminRolePaybackSystemRemote
+createProgramPartner(pID : String, pName : String, pDesc : String)
+updateProgramPartner(pID : String, pName : String, pDesc : String)
<<Remote>>
IBusinessRoleProgramPartnerRemote
+getMyServices() : Service [*]
+createService(pID : String, pName : String, pDesc : String, pPoints : int, pPointsIn : boolean)
+updateService(pID : String, pName : String, pDesc : String, pPoints : int, pPointsIn : boolean)
+getServiceByID(pID : String) : Service
+associateRoleEntity(pID : String)
entity
<<Entity>>
ProgramPartner
<<Id>>-programPartnerID : String
-programPartnerName : String
-programPartnerDescription : String
<<OneToMany>>-deliveredServices : Service [*]{ordered}
<<create>>+ProgramPartner()
<<create>>+ProgramPartner(pID : String, pName : String, pDesc : String)
Serializable
Serializable
<<Entity>>
Service
<<Id>>-serviceID : String
-serviceName : String
-seriveDescription : String
-points : int
-pointsIn : boolean
<<ManyToOne>>-owner : ProgramPartner
<<create>>+Service()
<<create>>+Service(pID : String, pName : String, pDesc : String, pPoints : int, pPointsIn : boolean)
ejb
<<Stateful>>
BusinessRoleProgramPartnerSFSB
<<EJB>>-app : ApplicationPaybackSystemSGSB
-roleProgramPartner : ProgramPartner
<<Stateful>>
AdminRolePaybackSystemSFSB
<<EJB>>-app : ApplicationPaybackSystemSGSB
<<Stateless>>
CommonRolePaybackSystemSLSB
<<EJB>>-app : ApplicationPaybackSystemSGSB
<<Singleton>>
<<Startup>>
<<LocalBean>>
ApplicationPaybackSystemSGSB
-programPartners : ProgramPartner [*]{ordered}
<<EJB>>-dataMapper : IDataPersistence
<<PostConstruct>>+initializeApp()
+createProgramPartner(pp : ProgramPartner)
+updateProgramParnter(pp : ProgramPartner)
+getAllProgramPartners() : ProgramPartner [*]
+getProgramPartnerByID(pID : String) : ProgramPartner
+getProgramPartnersHaveNoPointInServ() : ProgramPartner [*]
+getAllServices() : Service [*]
+getAllPointsOutService() : Service [*]
+getServicesOfAProgramPartner(pID : String) : Service [*]
Persistence
<<Stateless>>
DataPersistenceSLSB
<<PersistenceContext>>-em : EntityManager
<<Local>>
IDataPersistence
+persistProgramPartner(pp : ProgramPartner)
+updateProgramPartner(pp : ProgramPartner)
+queryAllProgramPartners() : ProgramPartner [*]

Fig. 5.12.: The transformed internal structure model targeted on the JEE platform

The complete structure of the generated JEE-application is illustrated in Figure 5.12, which is in fact the internal target model created by the *JEEStandardModelMapper*. Each of the classifiers in Figure 5.12 corresponds to a separate Java source file residing in its owning package. To obtain better readability, all the generated getters and setters are omitted. Compared to the platform independent design model in Figure 5.10, the transformed interfaces and classes conform to the JEE programming paradigm in that the data entities continue to be the *entity*-classes according to the JPA specification, whereas the role-based business operations are declared by the respective *remote*-interface embodying the behavior of that role and implemented in the *EJB*-class implementing that remote interface. Moreover, each layer exposes its defined behaviors only in the form of Java interfaces, which are represented always in green background color in Figure 5.12, the loose coupling among layers is achieved. The remote interfaces in *business object layer* can be *injected* into or obtained via JNDI mechanism by diverse front-end applications.

The actual MOCCA compilation flow ends in the phase of code generation for JEE applications. Thus, a default directory structure identical to the one for JSE is created to save the generated Java source files. Moreover, none of the diverse deployment descriptors, e.g., the *persistence.xml*, can be generated by MOCCA. With the future development of platform specific *deployment generator*, a sophisticated directory structure, e.g., compatible to Apache Maven tool as well as the Maven *pom.xml* file will be generated to deploy JEE applications. For the time being, the generated Java packages including their source files will be imported into an IDE like *NetBeans* [Ora13b] to further compile and deploy the application onto a JEE application server like *GlassFish* [Ora13a].

```
 1 @Lock(value = LockType.READ)
 2 public List<ProgramPartner> getPartnersHaveNoPointsInServ() {
 3    ArrayList<ProgramPartner> selectResult_0 = new ArrayList<ProgramPartner>();
 4    Iterator<ProgramPartner> itr_0 = this.getAllProgramPartners().iterator();
 5    while (itr_0.hasNext()) {
 6       ProgramPartner itrVar_0 = itr_0.next();
 7       boolean forAllResult_0 = true;
 8       Iterator<Service> itr_1 = itrVar_0.getDeliveredServices().iterator();
 9       while (itr_1.hasNext()) {
10          Service itrVar_1 = itr_1.next();
11             forAllResult_0 = forAllResult_0 && !itrVar_1.getPointsIn();
12       }
13       if (forAllResult_0) {
14            selectResult_0.add(itrVar_0);
15       }
16    }
17   return selectResult_0;
18 }
```

Listing 5.28: The generated Java code corresponding to the XOCL expression in Listing 5.19

The result of structural mapping is shown in Figure 5.12. As done for the JSE mapping in the previous section, three transformed Java methods are chosen to explore the benefit of using XOCL as action language. Listing 5.28 is the Java code corresponding to the XOCL expression in Listing 5.19 with *nested* usages of the *select()* and *forAll()* (lines 7 to 12) loop operations. It is worth noting that this operation is enhanced by the *«CommonRole»* stereotype in design model and such a distributed query operation is mapped as a *transaction*-operation with *read*-lock based on the EJB *container managed transaction*. The Java code in Listing 5.29 corresponds to the very compact XOCL expression in Listing 5.20, which is interpreted as the shorthand notation of the *collect()* loop operation. Because the association-end *deliveredServices* is a collection again, a nested *while*-loop (lines 9 to 12) is generated to *flatten* the final result collection. The Java code in Listing 5.30 implements the XOCL block expression in Listing 5.26. The Java code in lines 4 to 12 in Listing 5.30 corresponds with the invocation of the *any()* loop operation in line 6 in Listing 5.26. It is obvious that the "compression ratio" of XOCL imperative expressions is not as high as achieved for query expressions. However, the mapped target code is at least not more compact than the XOCL imperative counterparts.

```java
@Lock(value=LockType.READ)
public List<Service> getAllServices()
{
   ArrayList<Service> collectResult_0= new ArrayList<Service>();
    Iterator<ProgramPartner> itr_0 = this.getProgramPartners().iterator();
    while(itr_0.hasNext()){
         ProgramPartner itrVar_0 = itr_0.next();
         Iterator<Service> itr_1 = itrVar_0.getDeliveredServices().iterator();
         while(itr_1.hasNext()){
             Service itrVar_1 = itr_1.next();
             collectResult_0.add(itrVar_1);
         }
    }
    return collectResult_0;
}
```

Listing 5.29: The generated Java code corresponding to the XOCL expression in Listing 5.20

```java
public void updateProgramPartner(ProgramPartner pp)
{
    if(!this.getProgramPartners().contains(pp)){
       ProgramPartner anyResult_0 = null;
       Iterator<ProgramPartner> itr_0 = this.getProgramPartners().iterator();
       while (itr_0.hasNext()) {
           ProgramPartner itrVar_0 = itr_0.next();
           if (itrVar_0.getProgramPartnerID().equals(pp.getProgramPartnerID())){
               anyResult_0 = itrVar_0;
               break;
           }
       }
       ProgramPartner p = anyResult_0;
       p.setProgramPartnerName(pp.getProgramPartnerName());
       p.setProgramPartnerDesc(pp.getProgramPartnerDesc());
       p.setDeliveredServices(pp.getDeliveredServices());
    }
    this.dataMapper.updateProgramPartner(pp);
}
```

Listing 5.30: The generated Java code corresponding to the XOCL expression in Listing 5.26

The data base related operations can be enhanced by special stereotypes like *«DBQuery»*. A platform specific model mapper like *JEEStandardModelMapper* can process them and therefore generate implementation code with some default semantics automatically. Listing 5.31 shows the default Java implementation based on the JPA *query language* (line 5) and other JPA API for a *«DBQuery»*-operation. As shown in the listing, the sole required information in such a default JPA query is the entity type, here is the *ProgramPartner*, that can be inferred from the return type of the operation.

```java
 1 @Override
 2 public List<ProgramPartner> queryAllProgramPartners()
 3 {
 4     List<ProgramPartner> queryResult = null;
 5     String qs = "SELECT␣identVar␣FROM␣ProgramPartner␣identVar";
 6     try {
 7         queryResult = em.createQuery(qs).getResultList();
 8     } catch (Exception e) {
 9         e.printStackTrace(System.err);
10     }
11     return queryResult;
12 }
```

Listing 5.31: Generated Java implementation for PersistenceManager:: queryAllProgramPartners()

5.2.4. Transformation Result on the ABAP Target Platform

As a proof of the MDA philosophy that one PIM can be transformed into different PSMs, the same *Simple Payback System* design model can also be mapped onto the *SAP NetWeaver Application Server ABAP* target platform. It is not supposed that all the possible readers of this thesis are familiar to this platform. Hence, a short introduction is given, followed by highlighting the characteristics of this platform, which affect the working principle of the underlying ABAP model mapper as well as the code generator.

In short, the SAP NetWeaver is the technology platform, with which a much stronger separation between technology and application has achieved [KK07]. SAP NetWeaver offers an entire group of so-called *usage types*, among which *Application Server* (AS) can be used to develop and deploy applications within SAP NetWeaver. The Application Server ABAP (AS ABAP) is the part of the application server usage type in SAP NetWeaver with which applications can be programmed in ABAP. There is also Application Server Java (AS Java), which can be considered as SAP's implementation of the JEE standard to some extent. ABAP (Advanced Business Application Programming) is a programming language that was developed by SAP for developing commercial applications in the SAP environment. In the early days (i.e., the 1970s), ABAP stood for *Allgemeiner Berichts-Aufbereitungs Prozessor* (Generic Report Generation Processor). ABAP in this stage was implemented as a macro assembler under R/2, and as the name implies, was used exclusively for creating reports [KK07].

With decades of evolution, especially to support the object-oriented paradigm, the latest ABAP has become a full-fledged object-oriented programming language and is called ABAP Objects. However, unlike Java and C#, which were designed from beginning on for object orientation, obsolete non-OO language constructs still remain in the ABAP Objects for downward compatibility. This decision makes ABAP Objects an overstaffed language with more than 500 statements [FK08]. The reason for this is that ABAP has more than 200 million lines of production application code in use by SAP customers worldwide, and SAP guarantees its customers that this code will be executable under new versions of the language [KK07]. In this thesis, only the latest and recommended language constructs will be used to map an application design model.

As discussed in Chapter 4, to map the same PIM onto different target platforms, the according model mapper must be able to handle the platform-specific details. Before the transformation result is shown, important language characteristics, which must be handled by the ABAP model mapper, will be highlighted together with the mapping strategy to deal with these characteristics.

- There are special naming conventions and restrictions for ABAP development objects.

In ABAP, the names of all the development objects like *package*, *class*, etc, developed by a SAP partner must begin with a prefix reserved by SAP for that partner. Otherwise a customer must use "Y" or "Z" for the first letter in their development objects [KK07]. In this thesis, only local classes [1] will be generated by the ABAP code generator, which can be indicated by the prefix "lcl_". Besides the required prefix, other restrictions regarding naming conventions have also to be held. For example, the name of a class (local and global) is allowed to contain maximal 30 characters, whereas 16 characters are allowed for database tables.

Taking into account these conventions and restrictions, the *ABAPBasicModelMapper* implemented in this thesis renames a local class by prefixing it with "lcl_" and checks the length of the class name after renaming against the allowed 30. If longer than 30, the first 27 characters

[1]An ABAP local class is only visible in the program, in which it is defined, while a global class can be used in every program of the same AS ABAP. The global classes of an AS ABAP form its class library. Furthermore, local classes can be defined and implemented in ABAP editor, whereas the definition of global classes can only be generated with the integrated tool *Class Builder* by filling the relevant template parameters

will be preserved and suffixed with an auto-generated number, which avoids possible name conflict. The same strategy is applied to all the development objects, which have to be renamed according to the ABAP naming convention. For example, database tables will be prefixed with "YDBT_", etc. Moreover, ABAP is not case sensitive. Thus, the ABAP environment does not make difference between uppercase letters and lowercase letters.

- Methods cannot be overloaded in ABAP Objects.

Method overloading occurs if several methods have the same name but different parameters. In the design model shown in Figure 5.10 on page 131, two methods with the same name *createProgramPartner()* but different parameter list are defined in the class *PaybackSystem*. Method overloading is very useful in both modeling and implementation. Hence, mainstream OOPLs like Java and C# support method overloading so well, that overloaded methods in PIM can be mapped onto such platforms seamlessly. However, ABAP Objects does not allow method overloading. To overcome this restriction, the *ABAPBasicModelMapper* maintains a list of the overloaded methods in a design model class, within which only the first overloaded method preserves its name, the others are suffixed with "OLM*n*", with *n* an auto-generated number. If the original method name contains more than 26 characters, only the first 26 characters will be used to combine with the 4-digit suffix, such that the allowed 30 characters for naming methods cannot be exceeded.

- Concrete collection types with explicit name can be defined as local types within a class.

Recall from Listing 4.3 and its according explanation on page 86, a concrete XOCL collection such as *OrderedSet(Service)* for defining typed elements in design model can be mapped into Java as *ArrayList<Service>* to define local variables, properties or operation parameters. In both XOCL and Java, a concrete collection type is usually used anonymously. In ABAP Objects, the internal tables [KK07] [Kel05] [FK08] can be considered as the generic collection types. If they are specialized with concrete data type that will be collected, the resulted concrete collection types can be defined as *local types* with explicit names within the class, whose properties or operation parameters are defined using these concrete collection types. As an example, a concrete collection type called *ltab_lcl_Service* with *HASHED TABLE* as its raw type and the reference to the class *lcl_Service* as its content is defined as a local type by the *TYPES* statement in line 3 in Listing 5.32. After definition, the *ltab_lcl_Service* can be used to define property in line 8 and to declare return value for an operation as shown in line 5 in Listing 5.32.

```
1 CLASS lcl_ProgramPartner DEFINITION.
2   PUBLIC SECTION.
3     TYPES ltab_lcl_Service TYPE HASHED TABLE OF REF TO lcl_Service WITH UNIQUE KEY
          table_line.
4
5       METHODS getMyServices RETURNING value(r) TYPE ltab_lcl_Service.
6
7    PRIVATE SECTION.
8      DATA deliveredServices TYPE ltab_lcl_Service.
9 ENDCLASS.
```

Listing 5.32: Concrete collection type defined as local type in an ABAP local class.

- The functional methods cannot be used as operands in all situations.

According to ABAP, a *functional method* is a method that can have any number of input parameters and only one return value that is passed by value [KK07]. Functional methods are intended to be used in operand positions. However, in AS ABAP Release 7.0 [2], with which the generated ABAP code is tested, the functional methods can neither be used as parameter in another method call, nor as target operand for internal table related statements. To solve this problem, the *ABAPStandardXOCLMapper* always generates an intermediate local variable to save the return value of a functional method for the usage as operand in the both situations mentioned above. Listing 5.33 shows an example for nested method call. Using functional methods as operands in other not allowed positions just follows the same principle.

```
//a nested XOCL operation call
update obj1.opCall( obj2.nestedOpCall() );

//compiler generated ABAP solution
DATA ld_virtualcaller_0 TYPE returnTypeOfNestedOpCall.
ld_virtualcaller_0 = obj2->nestedOpCall( ).
obj1->opCall( ld_virtualcaller_0 ).
```

Listing 5.33: Mapping strategy for ABAP functional methods used as operands in not allowed positions.

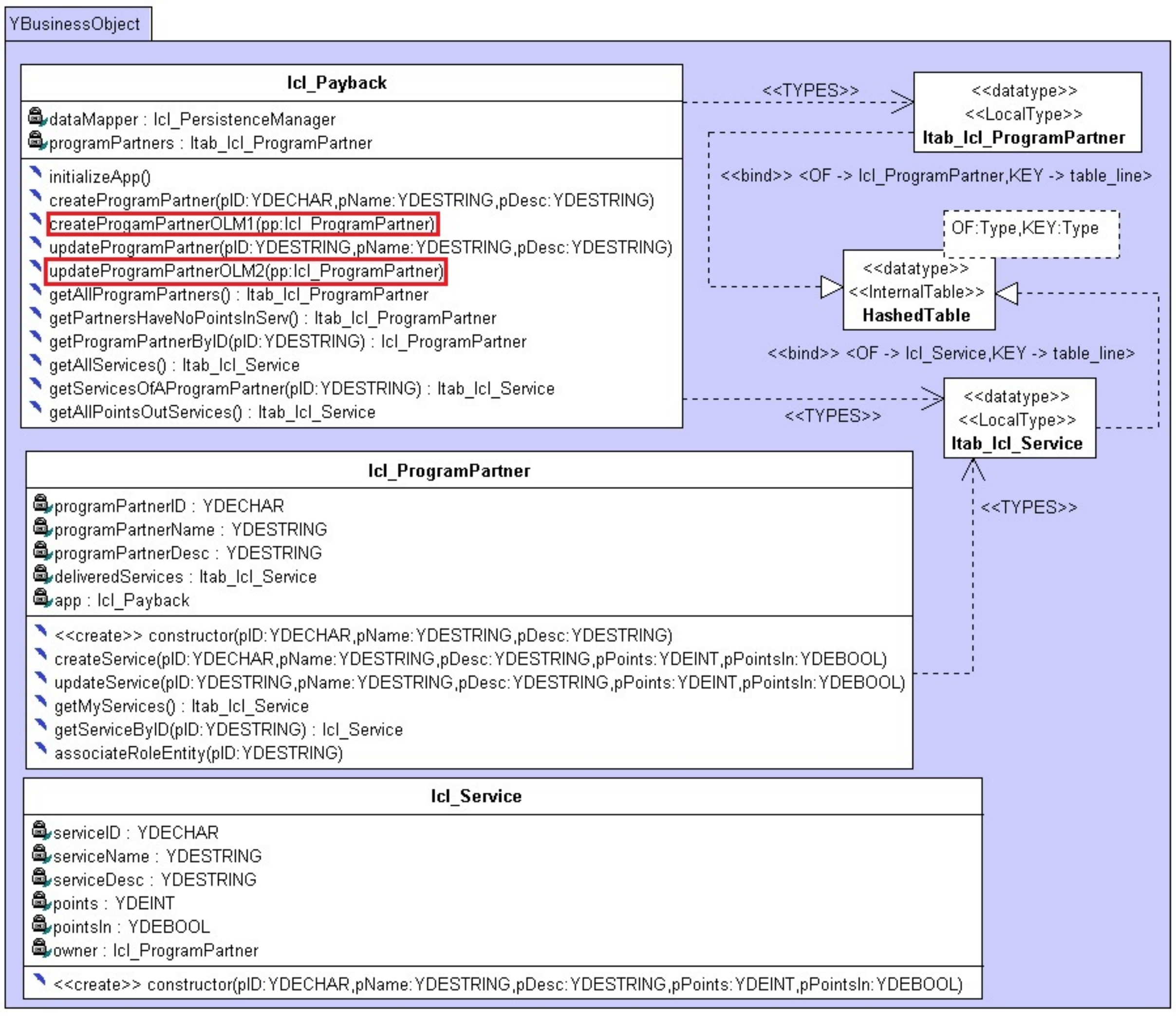

YBusinessObject
lcl_Payback
dataMapper : lcl_PersistenceManager
programPartners : Itab_lcl_ProgramPartner
initializeApp()
createProgramPartner(pID:YDECHAR,pName:YDESTRING,pDesc:YDESTRING)
createProgamPartnerOLM1(pp:lcl_ProgramPartner)
updateProgramPartner(pID:YDESTRING,pName:YDESTRING,pDesc:YDESTRING)
updateProgramPartnerOLM2(pp:lcl_ProgramPartner)
getAllProgramPartners() : Itab_lcl_ProgramPartner
getPartnersHaveNoPointsInServ() : Itab_lcl_ProgramPartner
getProgramPartnerByID(pID:YDESTRING) : lcl_ProgramPartner
getAllServices() : Itab_lcl_Service
getServicesOfAProgramPartner(pID:YDESTRING) : Itab_lcl_Service
getAllPointsOutServices() : Itab_lcl_Service
<<TYPES>>
<<datatype>>
<<LocalType>>
Itab_lcl_ProgramPartner
<<bind>> <OF -> lcl_ProgramPartner,KEY -> table_line>
OF:Type,KEY:Type
<<datatype>>
<<InternalTable>>
HashedTable
<<bind>> <OF -> lcl_Service,KEY -> table_line>
<<datatype>>
<<LocalType>>
Itab_lcl_Service
<<TYPES>>
<<TYPES>>
lcl_ProgramPartner
programPartnerID : YDECHAR
programPartnerName : YDESTRING
programPartnerDesc : YDESTRING
deliveredServices : Itab_lcl_Service
app : lcl_Payback
<<create>> constructor(pID:YDECHAR,pName:YDESTRING,pDesc:YDESTRING)
createService(pID:YDECHAR,pName:YDESTRING,pDesc:YDESTRING,pPoints:YDEINT,pPointsIn:YDEBOOL)
updateService(pID:YDESTRING,pName:YDESTRING,pDesc:YDESTRING,pPoints:YDEINT,pPointsIn:YDEBOOL)
getMyServices() : Itab_lcl_Service
getServiceByID(pID:YDESTRING) : lcl_Service
associateRoleEntity(pID:YDESTRING)
lcl_Service
serviceID : YDECHAR
serviceName : YDESTRING
serviceDesc : YDESTRING
points : YDEINT
pointsIn : YDEBOOL
owner : lcl_ProgramPartner
<<create>> constructor(pID:YDECHAR,pName:YDESTRING,pDesc:YDESTRING,pPoints:YDEINT,pPointsIn:YDEBOOL)

Fig. 5.13.: The transformed internal structure of the Business Object Layer in ABAP Objects

[2]As of the next release of AS ABAP, the functional methods can be used in almost all operand positions, where it is useful to do so [KK07].

- Class definition in ABAP Objects is separated into declaration and implementation part.

Different from Java but similar to C++, each class definition in ABAP Objects consists of a declaration part, which describes all the components of a class, and an implementation part, which implements the methods declared in a class. This syntax phenomena does not affect ABAP model mapper but the code generator, which must organize and assemble the transformed ABAP classes in correct order. In the *ABAPBasicCodeGenerator* implemented in this thesis, separate strings are maintained to represent class declaration and implementation respectively.

```
*-----------------------------------------------------------------*
*  CLASS lcl_PaybackSystem DEFINITION
*-----------------------------------------------------------------*
*  This ABAP class was generated by [Compiler Component ABAPBasicCodeGenerator]
*  at 8:27:23 on 10. June 2013.
*-----------------------------------------------------------------*
CLASS lcl_PaybackSystem DEFINITION .
   PUBLIC SECTION.
      TYPES ltab_lcl_ProgramPartner TYPE HASHED TABLE OF REF TO lcl_ProgramPartner WITH
           UNIQUE KEY table_line.
      TYPES ltab_lcl_Service TYPE HASHED TABLE OF REF TO lcl_Service WITH UNIQUE KEY
           table_line.

      METHODS initializeApp.

      METHODS getProgramPartners
                 RETURNING value(r) TYPE ltab_lcl_ProgramPartner.
      METHODS setProgramPartners IMPORTING p_programPartners TYPE
           ltab_lcl_ProgramPartner.

      METHODS createProgramPartner IMPORTING pID TYPE YDECHAR pName TYPE YDESTRING
           pDesc TYPE YDESTRING.

      METHODS createProgramPartnerOLM1 IMPORTING pp TYPE REF TO lcl_ProgramPartner.

      METHODS updateProgramPartner IMPORTING pID TYPE YDESTRING pName TYPE YDESTRING
           pDesc TYPE YDESTRING.

      METHODS updateProgramPartnerOLM2 IMPORTING pp TYPE REF TO lcl_ProgramPartner.

      METHODS getAllPointsOutServices
                 RETURNING value(r) TYPE ltab_lcl_Service.

      METHODS getAllProgramPartners
                 RETURNING value(r) TYPE ltab_lcl_ProgramPartner.

      METHODS getAllServices
                 RETURNING value(r) TYPE ltab_lcl_Service.

      METHODS getPartnersHaveNoPointsInServ
                 RETURNING value(r) TYPE ltab_lcl_ProgramPartner.

      METHODS getProgramPartnerByID IMPORTING pID TYPE YDESTRING
                 RETURNING value(r) TYPE REF TO lcl_ProgramPartner.

      METHODS getServicesOfAProgramPartner IMPORTING pID TYPE YDESTRING
                 RETURNING value(r) TYPE ltab_lcl_Service.

   PRIVATE SECTION.
      DATA dataMapper TYPE REF TO lcl_PersistenceManager.
      DATA programPartners TYPE ltab_lcl_ProgramPartner.
ENDCLASS.
```

Listing 5.34: The complete class declaration in ABAP generated from the design class PaybackSystem

Up to this point, the mapping result of the Business Object Layer of the *Simple Payback System* can be examined. Mapping persistence layer involves more complex transformation, which will be addressed later in this section. As shown in Figure 5.13, compared to the JEE structural model in Figure 5.12, there is no much structural change in the transformed ABAP target model. Because the generated ABAP classes will only be run within AS ABAP, which can be accessed via SAP GUI [KK07] on remote terminal computers. The classes have been renamed according to the naming convention discussed above. The renaming mechanism of overloaded methods have been highlighted by the rectangle with red frame color. Moreover, the generated getters and setters are omitted. It is to note that in ABAP target model, the generic internal table have been concretized with necessary information. In this example, two concrete collection types, namely, the *ltab_lcl_ProgramPartner*, and *ltab_lcl_Service* have been generated. The template parameter *OF* specifies the type, which should be saved by the current collection. The ABAP-specific pseudo type *table_line* is used for non-structured line type (here is the object reference), which turns the entire line into a key [KK07]. The *dependencies* stereotyped by *«TYPES»* indicate that the corresponding concrete collection types should be defined as local types in the depending classes as recalled from Listing 5.32.

To make the readers get more familiar with the *look-and-feel* of ABAP, the complete class declaration for the design class *PaybackSystem* emitted by the *ABAPBasicCodeGenerator* is shown in Listing 5.34. Noting that lines 1 to 6 are ABAP multiline comments, which contain some self-descriptive information generated by the ABAP code generator.

```
1  METHOD getPartnersHaveNoPointsInServ.
2     DATA ld_selectresult_0 TYPE ltab_lcl_ProgramPartner.
3     DATA ldr_itrvar_0 TYPE REF TO lcl_ProgramPartner.
4
5     DATA ld_virtualcaller_0 TYPE ltab_lcl_ProgramPartner.
6      ld_virtualcaller_0 = me->getAllProgramPartners( ).
7
8     LOOP AT ld_virtualcaller_0 INTO ldr_itrvar_0.
9        DATA ld_forallresult_0 TYPE YDEBOOL.
10        ld_forallresult_0 = 1.
11       DATA ldr_itrvar_1 TYPE REF TO lcl_Service.
12       DATA ld_virtualcaller_1 TYPE ltab_lcl_Service.
13        ld_virtualcaller_1 = ldr_itrvar_0->getDeliveredServices( ).
14
15       LOOP AT ld_virtualcaller_1 INTO ldr_itrvar_1.
16              DATA ld_virtualcaller_2 TYPE YDEBOOL.
17               ld_virtualcaller_2 = ldr_itrvar_1->getPointsIn( ).
18               IF ld_virtualcaller_2 = 1.
19                   ld_virtualcaller_2 = 0.
20               ELSE.
21                   ld_virtualcaller_2 = 1.
22               ENDIF.
23               IF ld_virtualcaller_2 = 0.
24                   ld_forallresult_0 = 0.
25                   EXIT.
26               ENDIF.
27       ENDLOOP
28
29        IF ld_forallresult_0 = 1.
30           INSERT ldr_itrvar_0 INTO TABLE ld_selectresult_0.
31        ENDIF.
32     ENDLOOP.
33     r = ld_selectresult_0.
34 ENDMETHOD.
```

Listing 5.35: The generated ABAP code corresponding to the XOCL expression in Listing 5.19

To explore the benefit of using XOCL as action language, the three methods identical to the ones chosen for JEE are used to show the generated ABAP code for method implementation. Listing 5.35 shows the ABAP code corresponding with the XOCL expression in Listing 5.19 with *nested forAll()* (lines 9 to 27) loop operation in a *select()* operation. It is to note that the lines

5 and 6 as well as lines 12 and 13 correspond with the solution for functional method calls as operand of *LOOP AT*-statement (lines 8 and 15), which iterates an internal table.

```
 1 METHOD getAllServices.
 2     DATA ld_collectResult_0 TYPE ltab_lcl_Service.
 3     DATA ldr_itrvar_0 TYPE REF TO lcl_ProgramPartner.
 4
 5     LOOP AT me->programPartners INTO ldr_itrvar_0.
 6         DATA ldr_itrvar_1 TYPE REF TO lcl_Service.
 7         DATA ld_virtualcaller_1 TYPE ltab_lcl_Service.
 8         ld_virtualcaller_1 = ldr_itrvar_0->getDeliveredServices( ).
 9
10         LOOP AT ld_virtualcaller_1 INTO ldr_itrvar_1.
11             INSERT ldr_itrvar_1 INTO TABLE ld_collectResult_0.
12         ENDLOOP.
13     ENDLOOP.
14     r = ld_collectResult_0.
15 ENDMETHOD.
```

Listing 5.36: The generated ABAP code corresponding to the XOCL expression in Listing 5.20

```
 1 METHOD updateProgramPartnerOLM2.
 2     DATA ld_virtualcaller_1 TYPE YDEBOOL.
 3     ld_virtualcaller_1 = 0.
 4     DATA ldr_itrvar_0 TYPE REF TO lcl_ProgramPartner.
 5
 6     READ TABLE me->programPartners FROM pp INTO ldr_itrvar_0.
 7     IF ldr_itrvar_0 IS INITIAL.
 8         ld_virtualcaller_1 = 1.
 9     ENDIF.
10
11     IF ld_virtualcaller_1 = 1.
12         DATA p TYPE REF TO lcl_ProgramPartner.
13         DATA ld_anyresult_0 TYPE REF TO lcl_ProgramPartner.
14         DATA ldr_itrvar_1 TYPE REF TO lcl_ProgramPartner.
15
16         LOOP AT me->programPartners INTO ldr_itrvar_1.
17             DATA ld_virtualcaller_3 TYPE YDECHAR.
18             ld_virtualcaller_3 = pp->getProgramPartnerID( ).
19             IF ldr_itrvar_1->getProgramPartnerID( ) = ld_virtualcaller_3.
20                 ld_anyresult_0 = ldr_itrvar_1.
21                 EXIT.
22             ENDIF.
23         ENDLOOP.
24         p = ld_anyresult_0.
25
26         DATA ld_virtualcaller_5 TYPE YDESTRING.
27         ld_virtualcaller_5 = pp->getProgramPartnerName( ).
28         p->setProgramPartnerName( ld_virtualcaller_5 ).
29
30         DATA ld_virtualcaller_7 TYPE YDESTRING.
31         ld_virtualcaller_7 = pp->getProgramPartnerDesc( ).
32         p->setProgramPartnerDesc( ld_virtualcaller_7 ).
33
34         DATA ld_virtualcaller_9 TYPE ltab_lcl_Service.
35         ld_virtualcaller_9 = pp->getDeliveredServices( ).
36         p->setDeliveredServices( ld_virtualcaller_9 ).
37     ENDIF.
38
39     me->dataMapper->updateProgramPartner( p = pp ).
40 ENDMETHOD.
```

Listing 5.37: The generated ABAP code corresponding to the XOCL expression in Listing 5.26

Listing 5.36 shows the generated ABAP code for the shorthand notation of the *collect()* loop operation in Listing 5.20. Because the association-end *deliveredServices* is a collection, a nested *LOOP AT* (line 10) is generated to flatten the final result collection. The ABAP code in Listing 5.37 implements the XOCL block expression in Listing 5.26. The ABAP code in lines 2 to 9 implements the code template of the XOCL *excludes()* operation, whose core logic are the *READ TABLE*-statement and the check agains *INITIAL*. The ABAP code in lines 13 to 23 in Listing 5.37 corresponds with the invocation of the *any()* loop operation in line 6 in in Listing 5.26. To deal with nested method call in ABAP, intermediate variables are generated in lines 26, 30 and 34, respectively.

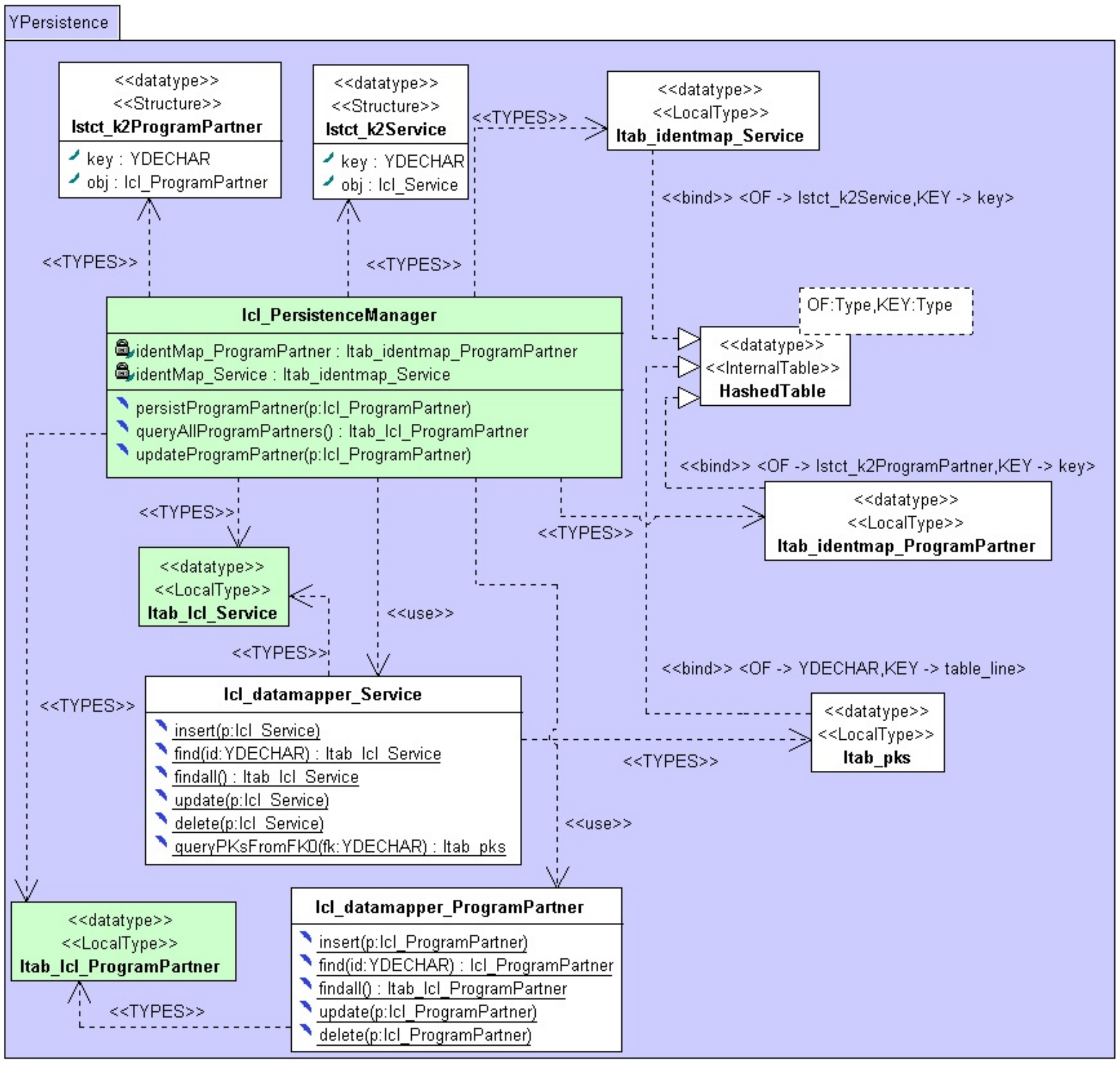

YPersistence
<<datatype>> <<Structure>> lstct_k2ProgramPartner
key : YDECHAR
obj : lcl_ProgramPartner
<<datatype>> <<Structure>> lstct_k2Service
key : YDECHAR
obj : lcl_Service
<<datatype>> <<LocalType>> ltab_identmap_Service
<<bind>> <OF -> lstct_k2Service,KEY -> key>
OF:Type,KEY:Type
<<datatype>> <<InternalTable>> HashedTable
<<bind>> <OF -> lstct_k2ProgramPartner,KEY -> key>
<<datatype>> <<LocalType>> ltab_identmap_ProgramPartner
lcl_PersistenceManager
identMap_ProgramPartner : ltab_identmap_ProgramPartner
identMap_Service : ltab_identmap_Service
persistProgramPartner(p:lcl_ProgramPartner)
queryAllProgramPartners() : ltab_lcl_ProgramPartner
updateProgramPartner(p:lcl_ProgramPartner)
<<datatype>> <<LocalType>> ltab_lcl_Service
<<use>>
<<bind>> <OF -> YDECHAR,KEY -> table_line>
lcl_datamapper_Service
insert(p:lcl_Service)
find(id:YDECHAR) : ltab_lcl_Service
findall() : ltab_lcl_Service
update(p:lcl_Service)
delete(p:lcl_Service)
queryPKsFromFK0(fk:YDECHAR) : ltab_pks
<<datatype>> <<LocalType>> ltab_pks
<<TYPES>>
<<datatype>> <<LocalType>> ltab_lcl_ProgramPartner
lcl_datamapper_ProgramPartner
insert(p:lcl_ProgramPartner)
find(id:YDECHAR) : lcl_ProgramPartner
findall() : ltab_lcl_ProgramPartner
update(p:lcl_ProgramPartner)
delete(p:lcl_ProgramPartner)

Fig. 5.14.: The transformed internal structure of the Persistence Layer in ABAP Objects

- There is no ready-to-use O/R mapping component based on the *data mapper* design pattern in ABAP

The *PersistenceManager* class residing in the *Persistence* layer as shown in Figure 5.10 is decorated with *«DataMapper»* stereotype. In a design model, it expresses clearly that the objects of a *«Persistence»* class should be saved into the underlying data repository via data mapper components [Fow+02] [Sch09]. In the case of relational database, the corresponding data mapper usually involves necessary SQL statements for CRUD operations. The *PersistenceManager* serves as a *facade* [Bal05] connecting to the underlying concrete data mappers. For a target platform like JEE, an O/R mapping component based on the *data mapper* design pattern is provided. In this case, transforming *PersistenceManager* and its involved methods is straightforward as shown in Listing 5.31. However, there is no ready-to-use O/R mapping component based on the *data mapper* design pattern in ABAP[3]. As addressed in [Bal05], implementing a general purpose O/R mapping framework based on data mapper pattern is very complicated. Because this framework should deal with any type of persistence classes that are not predictable to the framework.

To explore how well a model mapper can interpret the modeling elements with high-level semantics, such as stereotypes, and extract information from a UML model, the *ABAPBasic-ModelMapper* attempts to generate a *lightweight O/R mapping framework tailored only to the persistence classes in the current project* from the information provided in the design model. It is feasible because all the persistence classes are known in a design model and at the time of transforming them the relationships among persistence class, the database table as counterpart, as well as the generated data mapper class for that persistence class can be maintained and traced for further usage. On the other hand, ABAP involves a small number of statements called *Open SQL* [KK07], which make the database access and manipulation straightforward and efficient.

```
1 METHOD insert.
2     DATA wa TYPE YDBT_Service.
3     DATA fkobj0 TYPE REF TO lcl_ProgramPartner.
4      fkobj0 = p->getowner( ).
5
6     wa-owner = fkobj0->getprogramPartnerID( ).
7     wa-points = p->getpoints( ).
8     wa-pointsIn = p->getpointsIn( ).
9     wa-serviceDesc = p->getserviceDesc( ).
10    wa-serviceID = p->getserviceID( ).
11    wa-serviceName = p->getserviceName( ).
12
13    INSERT YDBT_Service FROM wa.
14 ENDMETHOD.
```

Listing 5.38: The generated ABAP code inserting a Service object into its database table

The generated persistence layer on the ABAP target platform is shown in Figure 5.14. The both collection types in green background color are the same ones as in Figure 5.13. Classes in white background color belong to the generated framework. It is easy to understand that for each *persistence* class in design model, a data mapper class involving all the necessary CRUD operations is generated. According to the current strategy, two methods, namely *find()* and *find-All()* are provided as the reading operation. The former brings back one object according to the *id*-parameter, whereas the latter queries all the objects and saves them in a collection. For an association-end such as *Service::owner*, which is resolved as *foreign key* relationship, an additional operation named *queryPKsFromFK()* will be generated in the corresponding data mapper class. This operation is suffixed with an auto-generated number to deal with the situation with more than one foreign key relationship. The *keyset* collection used by this operation as return value will also be generated. In this example, it is the *ltab_pks*. In the current version of ABAP model mapper, *combined foreign key* are not supported. As a code snippet of the entire generated data mapper classes, the *insert()* method of the *lcl_ datamapper_ Service* is given in Listing 5.38. It is to note that the database table *YDBT_ Service* can be used to define variable in line 2 directly. Such a variable is usually called *working area* in ABAP. Furthermore, the Open SQL statement *INSERT* in line 13 merges with other ABAP statements seamlessly. As presented in Figure 5.14, all the CRUD operations of both data mapper classes are underlined that marks them as *static*

[3]ABAP Objects provides indeed an O/R mapping framework, called Object Services [KK07]. However, the Object Services framework is based on the *active record* design pattern [Fow+02] [Sch09], which is considered not that flexible as data mapper.

methods. Hence, the generated data mapper classes provide only access to the respective data base table without recording any data transfer state. That is why they are connected to the the *facade* class, namely, *PersistenceManager* via *usage*-dependency.

```
*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *
*  CLASS lcl_PersistenceManager DEFINITION
*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *
*  This ABAP class was generated by [Compiler Component ABAPBasicCodeGenerator]
*  at 8:27:23 on 10. June 2013.
*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *
CLASS lcl_PersistenceManager DEFINITION.
   PUBLIC SECTION.
      TYPES: BEGIN OF lstct_k2ProgramPartner,
                   key TYPE YDECHAR,
                   obj TYPE  REF TO lcl_ProgramPartner,
               END OF lstct_k2ProgramPartner.

      TYPES: BEGIN OF lstct_k2Service,
                   key TYPE YDECHAR,
                   obj TYPE  REF TO lcl_Service,
               END OF lstct_k2Service.

      TYPES ltab_identmap_ProgramPartner
               TYPE HASHED TABLE OF lstct_k2ProgramPartner
               WITH UNIQUE KEY key.

      TYPES ltab_identmap_Service
               TYPE HASHED TABLE OF lstct_k2Service
               WITH UNIQUE KEY key.

      TYPES ltab_lcl_ProgramPartner
               TYPE HASHED TABLE OF REF TO lcl_ProgramPartner
               WITH UNIQUE KEY table_line.

      TYPES ltab_lcl_Service
               TYPE HASHED TABLE OF REF TO lcl_Service
               WITH UNIQUE KEY table_line.

      METHODS persistProgramPartner IMPORTING p TYPE REF TO lcl_ProgramPartner.

      METHODS queryAllProgramPartners
                   RETURNING value(r) TYPE ltab_lcl_ProgramPartner.

      METHODS updateProgramPartner IMPORTING p TYPE REF TO lcl_ProgramPartner.

   PRIVATE SECTION.
      DATA identMap_ProgramPartner TYPE ltab_identmap_ProgramPartner.
      DATA identMap_Service TYPE ltab_identmap_Service.
ENDCLASS.
```

Listing 5.39: The transformed PersistenceManager class in ABAP

[Bal05] summarizes the most essential concepts that a general O/R mapping framework based on data mapper pattern should implement. One of them concerns a cache, with which redundant database access can be avoided. According to [Fow+02] and [Sch09], this cache can be implemented based on the *identity map* design pattern, whose core structure is a *key to object* mapping for each object loaded in the previous database access. To support this principle, the *ABAPBasicModelMapper* generates for each *persistence* class an ABAP local structure and a hash table based local collection type maintaining this structure respectively. As shown in Figure 5.14, the local structure *lstct_k2ProgramPartner* and the local collection type *itab_identmap_ProgramPartner* are generated by the model mapper for the persistence class *ProgramPartner* in design model. After that the identity map for each persistence class can be defined as property in the *lcl_PersistenceManager* class as shown in Figure 5.14. The complete class declaration generated by the *ABAPBasicModelMapper* is shown in Listing 5.39.

5.2.5. Evaluation

In this experiment, the design model in Figure 5.10 on page 131 is mapped onto both JEE target platform and ABAP target platform. Both target platforms provide high-level services that allow software development to keep away from low-level cross-cutting concerns and plumbing issues on the one hand. On the other hand, for such application-server based target platforms, there are special programming models, which support developing software on them. As the JEE target model in Figure 5.12 as well as ABAP target model in both Figure 5.13 and 5.14 illustrate, the respective model mapper fills the gap between the application logic modeled in PIM and the required programming model on a target platform by creating necessary programming elements. The quantitative results of the structure mapping done by the respective model mapper are presented in Figure 5.15. It is clear that the application structure created in design model is more concise than its corresponding implementation on the both target platforms.

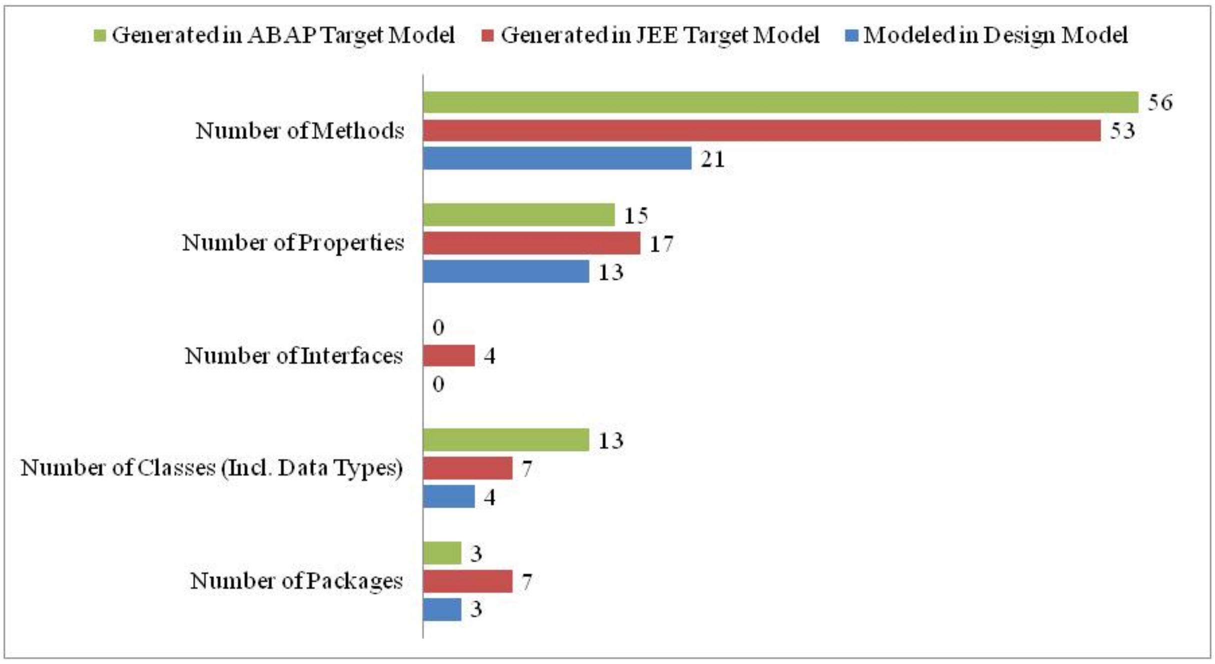


Fig. 5.15.: Evaluation of the structural model and its corresponding source code

To evaluate behavioral mapping, the operations of the design class *PaybackSystem* are classified into two groups, namely, the query operations and the non-query operations. As summarized in Figure 5.16, query operations can be specified using XOCL expressions very efficiently. For non-query operations, the corresponding Java implementations have almost the identical size to the XOCL expressions. However, due to the characteristics discussed in Section 5.2.4, the size of the generated ABAP code remains evidently larger than the XOCL counterpart.

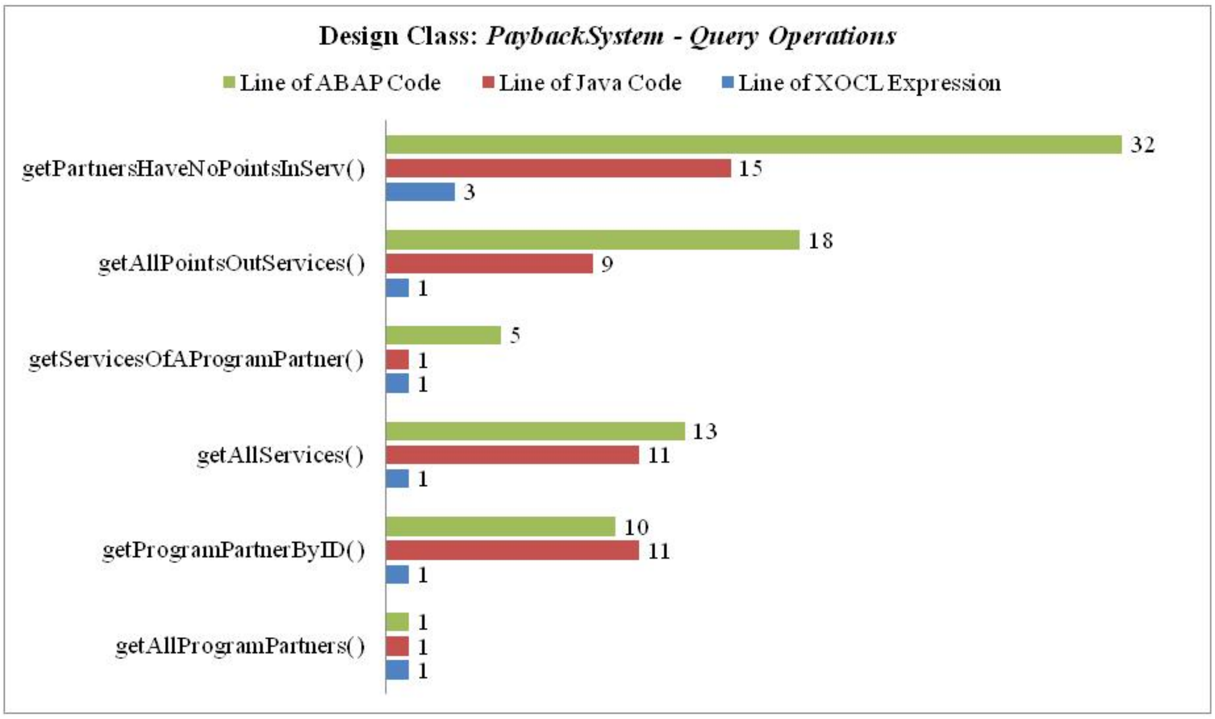


Fig. 5.16.: Evaluation of the behavioral specification and its corresponding source code for query operations

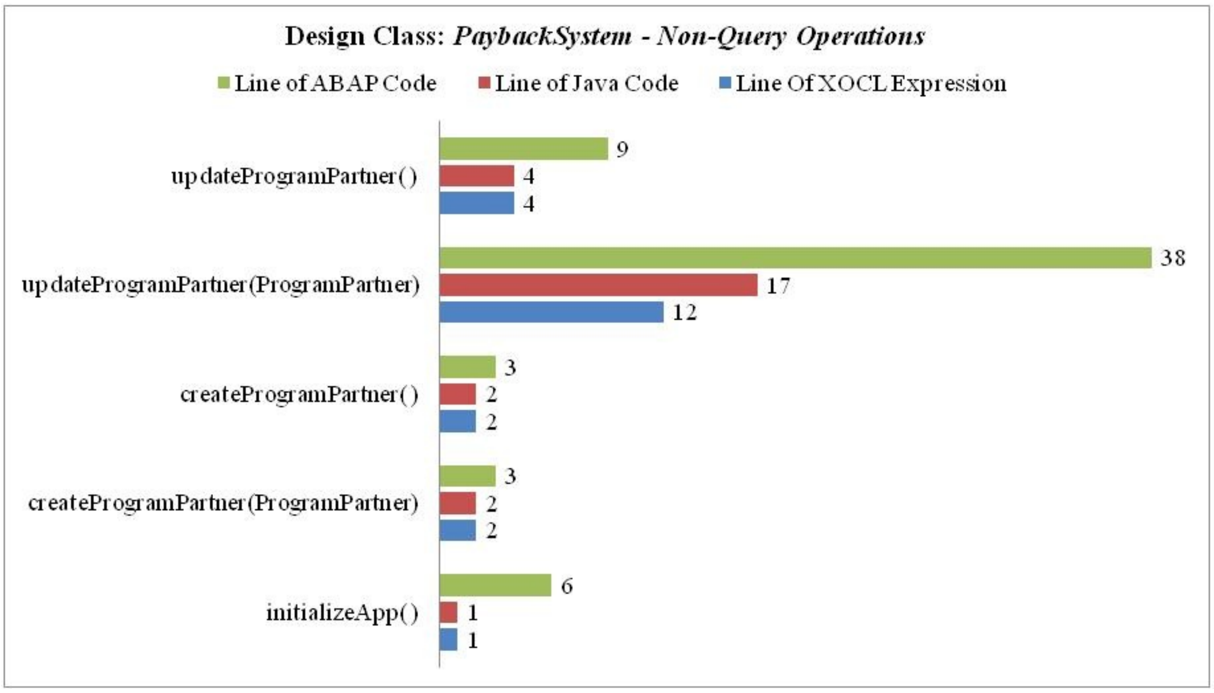


Fig. 5.17.: Evaluation of the behavioral specification and its corresponding source code for non-query operations

6. Conclusions

This thesis discussed the platform independent modeling together with the corresponding model transformation as well as code generation in the context of MDA. The theories and methodologies developed in this work found their origin in the early research of Dr. Föhlich [Frö07]. In the presented thesis the software architecture of MOCCA created by Dr. Föhlich has been reworked, refined and enhanced with new components supporting the methodologies developed in this thesis to prove that a *complete generation of executable application from its PIM created only by using UML/XOCL is feasible.*

One of the most remarkable characteristics of the constructive work done in this thesis is that both modeling and model transformation are practically realized with our own UML-related tool-set, which have been developed in diverse in-house research projects and commercial projects in cooperation with our industrial partner as summarized in Section 2.4. In the following, the overall achievements in both modeling and model transformation are concluded and the directions for future research are discussed.

Platform independent modeling

The UML is a powerful language for preparing models of object-oriented software. Such models can be used as documentation of existing software for their maintenance, and as source of new software to develop. In order to create precise models the OCL can be used to formulate constraints and specify complex query operations for the underlying UML model.

The aim of the MDA technology is the generation of program code for a certain platform based on a complete UML/OCL model. In order to gain benefit from this innovative technology, it is necessary to create the basic platform independent model (PIM) concisely, uniformly, completely and especially with low effort. While preparing such UML/OCL - PIMs, several serious problems have been recognized and in this thesis, the efficient solutions are proposed and realized to cope with all these modeling problems.

Similar to developing software in programming languages, certain fundamental modeling elements such as primitive types, collection types as well as their basic operations are ubiquitous in creating PIM of any kind of applications. To avoid repeated creating these fundamental building blocks at the time of modeling, a common modeling library called MOCCA design platform model (DPM) was conceived and its prototype has been built in the work of Dr. Fröhlich. In this thesis, the MOCCA DPM has been completely reworked to adopt OCL core data types and their predefined operations as its foundation. Further more, a novel IO facility has been added to the DPM, and even a platform independent GUI tool-kit including the most fundamental GUI elements has been created, too.

With introducing the GUI tool-kit, two accompanying issues were recognized and have been solved accordingly. One of them concerns describing both structural compositions and visual parameters of GUI elements. This thesis suggests using a normal UML-class of a window to model the contained GUI elements as their attributes, whereas the concrete parameter values dealing with composition relationships among the elements (diverse containers and their contained elements to arrange GUI elements within a window) and the visual properties are manipulated with a separate software component called Smart GUI-Editor binding to the current GUI window. The resulted parameter values are represented internally with XOCL expressions, which are platform independent as well, and can be modified and edited with relatively low effort, when the Smart GUI Editor is still in its infancy and not that powerful to deal with all kinds of visual manipulation of GUI elements. The other issue concerns connecting an event to its handling operation in a platform independent manner. In the presented thesis the OCL has been enhanced with a new kind of expression called XOCL-event-expression, which allows to model the registration of handling methods to the source event in a more compact, well understandable, and uniform manner independent of any potential target platform. This new approach simplifies the class diagrams strongly without losing completeness [LS10].

It is clear that the UML class diagram, which is used to model structure of an application in this thesis, contributes little to compress the structural information of the underlying application, but only shifts representing application structure from the code into its visual counterpart. To compress the structural model, or in other words, to enhance the information density in class diagram without losing application semantics, the UML stereotypes are used to enhance model elements in class diagram with additional semantics. The stereotypes created in this thesis support domain specific modeling as well as several well defined design patterns. These stereotypes are involved in a profile, which has been integrated into the MOCCA DPM.

The last issue in terms of creating complete PIM is about modeling behavior both concisely and compactly. The proposed solution in the presented thesis is a restricted extension of the OCL. The expressive, declarative OCL has been upgraded into a full-fledged action language, which is called XOCL. With XOCL, complex query operation can be specified as usual as with OCL body-expressions, whereas non-query operations with complex control flows can also be specified using the extended imperative language constructs.

As the future work to support modeling wider spectrum of application category in a platform independent manner, the current MOCCA DPM has to be extended with more building blocks, e.g., to support 2D/3D graphics, to deal with network communication, etc. Even for the OCL-based primitive and collection types adopted by the current DPM, extensions can be made to exploit the power of the underlying target platform. For example, the most modern OOPL target platforms support *regular expression* based string manipulations, which are very useful in diverse applications. However, the classical OCL String does not support such functions, which can be involved in the next version of DPM in the *XOCLString*.

For the GUI tool-kit, more GUI elements must be adopted and the Smart GUI Editor must be further developed to support the convenient visual manipulation of GUI elements. Adopting a new platform independent GUI element into the DPM GUI tool-kit is not that straightforward. Because such a GUI element represents usually a common abstraction of several different GUI controls (e.g., *JButton* in Swing, *QButton* in Qt as well as *Button* in WPF) with similar functions on different target platforms. Things get even more difficult, when within the same language platform there are different GUI frameworks used for different application domains or even for the same application domain. Typical examples are Swing/JavaFX [Dea11] for desktop applications developed in Java, GWT [Goo13]/JSF [Ora11b] for web-applications in the Java world, and JME/Android API for mobile applications; whereas Windows Form/WPF for desktop applications running on Windows, ASP.Net [Tro07] for Web-application; Qt for traditional C++ desktop applications, etc. However, with years of evolution, a much clearer trend has emerged that new GUI technology coming with an OOPL platform appears to streamline the creation of user interfaces for diverse targets like desktop, web and mobile device [Dea11] [Mac12] [Qt13]. This fact leads to reduce the cross-comparing among GUI elements and ease the abstraction of GUI elements.

To achieve more compact structural modeling, additional stereotypes can be added to hide cross-cutting concerns and plumbings. For behavioral modeling, XOCL shows its value as a full-fledged action language with very efficient expressions to specify query operation and bind events with their handling operations on the one hand. On the other hand using XOCL to better and more compact model imperative application logic remains a challenging research objective.

Model transformation and code generation

Model transformation maps an application-specific design model into functionally equivalent target models involving target platform specific services and peculiarities. The both kinds of models are internally stored with the identical meta model implementation to the one of UML 2 Designer, such that the PIM created in UML 2 Designer can be processed by MOCCA seamlessly. For each potential target platform that can be used to map a design model, the target platform model (TPM), a model mapper with dedicated interface requirements as well as an XML-based mapping configuration file defining elementary mapping rules between DPM and TPM modeling elements have to be provided. The model mapper for a special target platform embodies the best practices of realizing an application based on the services and architecture of that platform.

The current version of MOCCA supports mapping traditional desktop applications onto the Java Standard Edition target platform, mapping both the business object layer and the persistence layer of a three-layered business application onto the Java Enterprise Edition and onto SAP NetWeaver Application Server ABAP target platform with certain constraints and limitation. Further more, the attempt to map MES applications onto the DVDL target platform [Dör13] is also given and the primary experimental results proving the overall feasibility are optimistic.

In the subsequent development of MOCCA, corresponding model mappers targeting to more platforms will be developed, especially for the well-known .Net platform. Moreover, within the same target platform, more sophisticated model mappers can be added by inheriting basic or standard model mapper for that platform to reflect state-of-the-art evolution within that platform. Further more, the XML-based mapping configuration file can also be extended to support more sophisticated DPM to TPM mappings. The current XOCL mappers for different target languages implement only the most fundamental features to generate target codes correctly. There is much room to optimize the generated code as well as to exploit the most current language extension on certain target platform, like LINQ [Tro07] on .Net platform.

Another serious issue in terms of future development of MOCCA concerns generating deployment for an application to complete the entire automation process of application development. To generate deployment, necessary information has to be either involved into the design model or fed into the potential *deployment generator* via separate configurations. Such considerations remain as challenging research topics in the future development.

A. The XOCL Grammar Rules

In this appendix, all the XOCL grammar rules are given in *Backus-Naur Form* (BNF), but in the form required by GOLD Parsing System [Coo13]. As addressed in Section 3.4, XOCL is developed as the action language to model operation implementation in a platform independent design model. Based on the study in [Lia08], the sole language construct in the original OCL, which is appropriate to specify operation behavior is the OCL *body*-expression. Hence, the XOCL is defined with all the OCL language constructs in terms of *body*-expression as well as the imperative extensions, which make XOCL full-fledged action language. Moreover, the original OCL *let*-expression [WK03] [OMG10a] is removed, because its semantics can be replaced by the extended XOCL language construct for defining local variables.

1. <XOCL> ::= <BodyExpCS> | <BlockExpCS>

2. <BodyExpCS> ::= ’**body**’ ’**:**’ <OCLExpCS>

3. <BlockExpCS> ::= ’**begin**’ <ImperativeExpListCS> ’**end**’
   | ’**begin**’ ’**end**’

4. <ImperativeExpListCS> ::= <ImperativeExpListCS> <ImperativeExpCS>
   | <ImperativeExpCS>

5. <ImperativeExpCS> ::= <WhileExpCS>
   | <IfExpCS>
   | <OCLVarDeclarationCS> ’**;**’
   | <EventExpCS> ’**;**’
   | <AssignExpCS> ’**;**’
   | <DestroyObjectExpCS> ’**;**’
   | <ReplyExpCS> ’**;**’
   | <NonQueryFeatureCallExpCS> ’**;**’

6. <EventExpCS> ::= ’**event**’ ’**:**’ <OCLFeatureCallExpCS> ’∼’ <OCLFeatureCallExpCS>

7. <WhileExpCS> ::= ’**while**’ <OCLExpCS> <BlockExpCS> ’**endwhile**’

8. <IfExpCS> ::= ’**if**’ <OCLExpCS> ’**then**’ <BlockExpCS> ’**endif**’
   | ’**if**’ <OCLExpCS> ’**then**’ <BlockExpCS> ’**else**’ <BlockExpCS> ’**endif**’
   | ’**if**’ <OCLExpCS> ’**then**’ <OCLExpCS> ’**else**’ <OCLExpCS> ’**endif**’

9. <AssignExpCS> ::= ’**update**’ <OCLFeatureCallExpCS> ’=’ <OCLExpCS>
   | ’**update**’ <OCLFeatureCallExpCS> ’=’ <CreateObjectExpCS>

10. <NonQueryFeatureCallExpCS> ::= ’**update**’ <OCLFeatureCallExpCS>

11. <OCLVarDeclarationCS> ::= **ID** '**:**' <OCLTypeExpCS> '=' <OCLExpCS>
| **ID** '**:**' <OCLTypeExpCS> '=' <CreateObjectExpCS>
| **ID** '**:**' <OCLTypeExpCS>
| **ID**

12. <CreateObjectExpCS> ::= '**new**' <OCLFullNameExpCS>

13. <DestroyObjectExpCS> ::= '**delete**' <OCLFeatureCallExpCS>

14. <ReplyExpCS> ::= '**return**' **ID**

15. <OCLExpCS> ::= <OCLImpliesExpCS>

16. <OCLImpliesExpCS> ::= <OCLImpliesExpCS> '**implies**' <OCLLogicalExpCS>
| <OCLLogicalExpCS>

17. <OCLLogicalExpCS> ::= <OCLLogicalExpCS> '**and**' <OCLComparisonExpCS>
| <OCLLogicalExpCS> '**or**' <OCLComparisonExpCS>
| <OCLLogicalExpCS> '**xor**' <OCLComparisonExpCS>
| <OCLComparisonExpCS>

18. <OCLComparisonExpCS> ::= <OCLComparisonExpCS> '=' <OCLAdditiveExpCS>
| <OCLComparisonExpCS> '=' '**OclVoid**'
| <OCLComparisonExpCS> '<>' <OCLAdditiveExpCS>
| <OCLComparisonExpCS> '<>' '**OclVoid**'
| <OCLComparisonExpCS> '<=' <OCLAdditiveExpCS>
| <OCLComparisonExpCS> '>=' <OCLAdditiveExpCS>
| <OCLComparisonExpCS> '<' <OCLAdditiveExpCS>
| <OCLComparisonExpCS> '>' <OCLAdditiveExpCS>
| <OCLAdditiveExpCS>

19.<OCLAdditiveExpCS> ::= <OCLAdditiveExpCS> '+' <OCLMultExpCS>
| <OCLAdditiveExpCS> '**-**' <OCLMultExpCS>
| <OCLMultExpCS>

20. <OCLMultExpCS> ::= <OCLMultExpCS> '*****' <OCLUnaryExpCS>
| <OCLMultExpCS> '/' <OCLUnaryExpCS>
| <OCLUnaryExpCS>

21. <OCLUnaryExpCS> ::= '**not**' <OCLPrimitiveExpCS>
| '**-**' <OCLPrimitiveExpCS>
| <OCLPrimitiveExpCS>

22. <OCLPrimitiveExpCS> ::= <OCLLiteralExpCS>
| <OCLFeatureCallExpCS>

23. <OCLFeatureCallExpCS> ::= <OCLFeatureCallExpCS> '**->**' <OCLCollectionOpExpCS>
| <OCLFeatureCallExpCS> '**.**' <OCLFullNameExpCS>
| <OCLFullNameExpCS>

| '**self**'
| '(' <OCLExpCS> ')'

24. <OCLFullNameExpCS> ::= <OCLFullNameExpCS> '**::**' **ID** '(' <OCLArgumentsExpCS> ')'
| <OCLFullNameExpCS> '**::**' **ID**
| **ID** '(' <OCLArgumentsExpCS> ')'
| **ID**

25. <OCLCollectionOpExpCS> ::= <OCLNonloopOpExpCS>
| <OCLLoopOpExpCS>
| <OCLIterateExpCS>

26. <OCLNonloopOpExpCS> ::= **ID** '(' <OCLExpCS> '**,**' <OCLExpCS> ')'
| **ID** '(' <OCLExpCS> ')'
| **ID** '(' ')'

27. <OCLLoopOpExpCS> ::= '**forAll**' '(' <OCLLoopBodyExpCS> ')'
| '**exist**' '(' <OCLLoopBodyExpCS> ')'
| '**select**' '(' <OCLLoopBodyExpCS> ')'
| '**any**' '(' <OCLLoopBodyExpCS> ')'
| '**isUnique**' '(' <OCLLoopBodyExpCS> ')'
| '**one**' '(' <OCLLoopBodyExpCS> ')'
| '**reject**' '(' <OCLLoopBodyExpCS> ')'
| '**collect**' '(' <OCLLoopBodyExpCS> ')'
| '**sortedBy**' '(' <OCLLoopBodyExpCS> ')'
| '**collectNested**' '(' <OCLLoopBodyExpCS> ')'

28. <OCLIterateExpCS> ::= '**iterate**' '(' <OCLVarDeclarationCS> '**;**' <OCLVarDeclarationCS> '|'
<OCLExpCS> ')'
| '**iterate**' '(' <OCLVarDeclarationCS> '|' <BlockExpCS> ')'

29. <OCLLoopBodyExpCS> ::= <OCLVarDeclarationCS> '**,**' <OCLVarDeclarationCS> '|'
<OCLExpCS>
| <OCLVarDeclarationCS> '|' <OCLExpCS>
| <OCLExpCS>

30. <OCLArgumentsExpCS> ::= <OCLArgumentsExpCS> '**,**' <OCLExpCS>
| <OCLExpCS>
|

31. <OCLTypeExpCS> ::= <OCLPrimitiveTypeExpCS>
| <CollectionTypeExpCS>
| <OCLFullNameExpCS>

32. <OCLPrimitiveTypeExpCS> ::= '**Integer**'
| '**Real**'
| '**String**'
| '**Boolean**'

```
33. <CollectionTypeExpCS> ::= 'Set' '(' <OCLTypeExpCS> ')'
                            | 'Bag' '(' <OCLTypeExpCS> ')'
                            | 'OrderedSet' '(' <OCLTypeExpCS> ')'
                            | 'Sequence' '(' <OCLTypeExpCS> ')'

34. <OCLLiteralExpCS> ::= <OCLPrimitiveLiteralExpCS>
                        | <OCLCollectionLiteralExpCS>

35. <OCLPrimitiveLiteralExpCS> ::= StringLiteral
                                 | IntegerLiteral
                                 | RealLiteral
                                 | 'true'
                                 | 'false'

36. <OCLCollectionLiteralExpCS> ::= 'Set' '{' <OCLCollInitializerExpCS> '}'
                                  | 'Bag' '{' <OCLCollInitializerExpCS> '}'
                                  | 'OrderedSet' '{' <OCLCollInitializerExpCS> '}'
                                  | 'Sequence' '{' <OCLCollInitializerExpCS> '}'

37. <OCLCollInitializerExpCS> ::= <OCLCollInitializerExpCS> ',' <OCLCollInitPartsCS>
                                | <OCLCollInitPartsCS>
                                |

38. <OCLCollInitPartsCS> ::= <OCLPrimitiveLiteralExpCS>
                           | IntegerLiteral '..' IntegerLiteral
```

Books

[Aho+08] Aho, A., Lam, M., Sethi, R., and Ullman, J. *Compilers Principles, Techniques, & Tools*. 2nd ed. Pearson Education, 2008.

[Bal05] Balzert, H. *Lehrbuch der Objektmodellierung*. 2nd ed. Spektrum Akademischer Verlag, 2005.

[Bar05] Barker, J. *Beginning Java Objects: From Concepts to Code*. 2nd ed. Apress, 2005.

[Dea11] Dea, C. *JavaFX 2.0 Introduction by Example*. Apress, 2011.

[FK08] Färber, G. and Kirchner, J. *ABAP - Grundkurs*. 4th ed. Galileo Press, 2008.

[FQA12] Fawcett, J., Quin, L., and Ayers, D. *Beginning XML*. 5th ed. John Wiley & Sons, Inc, 2012.

[Fow+02] Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R., and Stafford, R. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002.

[FF04] Freeman, Eric and Freeman, Elisabeth. *Head First Design Patterns*. O'Reilly, 2004.

[Gam+95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gon10] Goncalves, A. *Beginning Java^{TM} EE 6 Platform with GlassFish^{TM} 3*. 2nd ed. Apress, 2010.

[HC08] Horstmann, C.S. and Cornell, G. *Core Java*. 8th ed. Prentice Hall, 2008.

[Kel05] Keller, H. *The Official ABAP Reference*. Galileo Press, 2005.

[KK07] Keller, H. and Krüger, S. *ABAP Objects*. 2nd ed. Galileo Press, 2007.

[LR09] Lahres, B. and Rayman, G. *Objektorientierte Programmierung*. 2nd ed. Galileo Computing, 2009.

[LB10] Lee, A. and Burke, B. *Enterprise JavaBeans 3.1*. 6th ed. O'Reilly, 2010.

[Mac12] MacDonald, M. *Pro WPF 4.5 in C#*. 4th ed. Apress, 2012.

[Ora11b] Oracle. *The Java EE 6 Tutorial*. Oracle, 2011.

[Pen03] Pender, T. *UML Bible*. Wiley Publishing, Inc., 2003.

[Rum11] Rumpe, B. *Modellierung mit UML*. 2nd ed. Springer, 2011.

[RQZ07] Rupp, C., Queins, S., and Zengler, B. *UML 2 Glasklar*. 3rd ed. Carl Hanser Verlag, 2007.

[Sch09] Schmidt, S. *PHP Design Patterns*. 2nd ed. O'Reilly, 2009.

[Sie09a] Siegfried, N. *QVT - Operational Mappings*. Springer, 2009.

[Sie09b] Siegfried, N. *QVT - Relations Language*. Springer, 2009.

[Som12] Sommerville, I. *Software Engineering*. 9th ed. Pearson Deutschland GmbH, 2012.

[Ste+09] Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. *EMF Eclipse Modeling Framework*. 2nd ed. Addison-Wesley, 2009.

[Tro07] Troelsen, A. *Pro C# 2008 and the .Net 3.5 Platform*. 4th ed. Apress, 2007.

[WK03] Warmer, J. and Kleppe, A. *The Object Constraint Language: Getting Your Models Ready for MDA*. 2nd ed. Addison Wesley, 2003.

[WKB03] Warmer, J., Kleppe, A., and Bast, W. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison Wesley, 2003.

[WC05] Wilton, P. and Colby, J. *Beginning SQL*. Wiley Publishing, Inc, 2005.

Open Standards and Online Documentations

[Acc13] Acceleo. *The Acceleo Project.* http://www.acceleo.org/pages/home/en, 2013.

[ANT13] ANT. *The Apache ANT Project.* http://ant.apache.org, 2013.

[ATL13] ATL. *The ATL Plug-in for Eclipse.* http://www.eclipse.org/atl/, 2013.

[Coo13] Cook, D. *Gold Parsing System Online Documentation.* http://goldparser.org, 2013.

[Ecl13] Eclipse. *The official Website of Eclipse Project.* http://www.eclipse.org/, 2013.

[Goo13] Google. *Google Web Toolkit.* https://developers.google.com/web-toolkit/, 2013.

[Haw13] Hawkins, M. *Gold Parsing System Java Engine.* http://goldparser.org/engine/1/java/doc/index.html, 2013.

[IBM12] IBM. *The Ecore API Online Documentation.* http://download.eclipse.org/modeling/emf/emf/javadoc/2.8.0/org/eclipse/emf/ecore/package-summary.html, 2012.

[JKU13] JKU. *The Compiler Generator Coco/R Online Documentation.* http://www.ssw.uni-linz.ac.at/coco/, 2013.

[Mav13] Maven. *The Apache MAVEN Project.* http://maven.apache.org/, 2013.

[OMG04] OMG. *Enterprise Collaboration Architecture (ECA) Specification.* www.omg.org/spec/EDOC/1.0/PDF/, 2004.

[OMG03] OMG. *MDA Guide Version 1.0.1.* www.omg.org/cgi-bin/doc?omg/03-06-01, 2003.

[OMG08] OMG. *MOF Model to Text Transformation Language, v1.0.* http://www.omg.org/spec/MOFM2T/1.0/PDF/, 2008.

[OMG10a] OMG. *Object Constraint Language.* www.omg.org/spec/OCL/2.2/PDF, 2010.

[OMG11a] OMG. *OMG Meta Object Facility (MOF) Core Specification.* www.omg.org/spec/MOF/2.4.1/PDF, 2011.

[OMG10b] OMG. *OMG Unified Modeling Language: Superstructure Specification.* www.omg.org/spec/UML/2.3/Superstructure/PDF, 2010.

[OMG11b] OMG. *Query/View/Transformation, V1.1.* www.omg.org/spec/QVT/1.1/PDF, 2011.

[Ora11a] Oracle. *Java Platform, Enterprise Edition 6 API Specification.* docs.oracle.com/javaee/6/api/, 2011.

[Ora12] Oracle. *Java Platform, Standard Edition 7 API Specification.* docs.oracle.com/javase/7/docs/api/, 2012.

[Ora13a] Oracle. *The official Website of GlassFish Project.* https://glassfish.java.net, 2013.

[Ora13b] Oracle. *The official Website of NetBeans Project.* https://netbeans.org, 2013.

[Qt13] Qt. *The Qt Official Website.* qt.digia.com, 2013.

[Spr13] Spring. *Spring Framework API Documentation.* http://static.springsource.org/spring/docs/3.1.x/javadoc-api/, 2013.

Other References

[And+99] Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H., Kuske, S., Plump, D., Schürr, A., et al. "Graph transformation for specification and programming". In: *Science of Computer Programming* Vol 34 issue 1 (1999), pp. 1–54.

[Apr13] Apromace. *Homepage of Apromace data systems GmbH.* www.apromace.com. 2013.

[BAI09a] BAIHB. *Modulhandbuch für den Bachelorstudiengang Angewandte Informatik.* http://tu-freiberg.de/zuv/service/pdf/modulhandbuecher/fakult1/z_ba_ainfo.pdf. 2009.

[BAI09b] BAIPO. *Prüfungs- und Studienordnung für den Bachelorstudiengang Angewandte Informatik.* http://tu-freiberg.de/zuv/service/pdf/studordg/2009/bai_po_so_ausfertigung.pdf. 2009.

[Bla04] Blankenhorn, K. *A UML Profile for GUI Layout.* Master Thesis. University of Applied Sciences Furtwangen. 2004.

[Bro+06] Broy, M., Crane, M., Dingel, J., Hartman, A., Rumpe, B., and Selic, B. "2nd UML 2 Semantics Symposium: Formal Semantics for UML". In: *MoDELS 06 Proceedings of the 2006 international conference on Models in software engineering.* Springer, 2006, pp. 318–323.

[CH03] Czarnecki, K. and Helsen, S. "Classification of Model Transformation Approaches". In: *OOPSLA 2003 Workshop on Generative Techniques in the context of Model Driven Architecture.* Oct. 2003.

[Dal13] Dalbey, John. *Pseudo code standard.* `http://users.csc.calpoly.edu/~jdalbey/SWE/pdl_std.html`. 2013.

[Did10] Didenko, I. *Reverse-Engineering Tool für die Generierung von Sequenzdiagramen aus C++ Projekten.* Bachelor Thesis. Freiberg University of Mining and Technology. 2010.

[DY03] Dong, J. and Yang, S. "Visualizing Design Patterns With A UML Profile". In: *Human Centric Computing Languages and Environments.* IEEE Symposium. 2003.

[Dör13] Döring, C. *Generieren DVDL-Code aus UML/XOCL-Anwendungsmodellen.* Master Thesis. Freiberg University of Mining and Technology. 2013.

[Dör10] Döring, C. *Smart GUI Editor.* Bachelor Thesis. Freiberg University of Mining and Technology. 2010.

[EPE06] Ehrig, H., Prange, U., and Ehrig, K. *Tutorial on Fundamentals of Algebraic Graph Transformation.* www.cs.le.ac.uk/events/segravis/material/Ehrig-Tutorial4.pdf. 2006.

[EPT04] Ehrig, H., Prange, U., and Taentzer, G. "Fundamental Theory for Typed Attributed Graph Transformation". In: *Graph Transformation, Proceedings of ICGT 2004.* Vol. 3256. Springer, 2004.

[Fow08] Fowler, M. *Domain Specific Language.* http://martinfowler.com/bliki/DomainSpecificLanguage.html. 2008.

[Frö07] Fröhlich, D. “Object-Oriented Development for Reconfiguralbe Architectures”. PhD thesis. `http://www.qucosa.de/fileadmin/data/qucosa/documents/2209/InformatikFrXXhlichDominik80246.pdf`, 2007.

[GMO09] Grønmo, R., Mølle-Pederson, B., and Olsen, G. “Comparison of Three Model Transformation Languages”. In: *Proceeding of the 5th European Conference on Model Driven Architecture: Foundations and Applications*. Springer, 2009, pp. 2–17.

[GLO09] Guerra, E., Lara, J., and Orejas, F. “Pattern-Based Model-to-Model Transformation: Handling Attribute Conditions”. In: *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations, ICMT’09*. Springer, 2009, pp. 83–99.

[HP04] Haustein, S. and Pleumann, J. “OCL as Expression Language in an Action Semantics Surface Language”. In: *OCL and Model Driven Engineering, UML 2004 Conference*. University of Kent, 2004, pp. 99–113.

[HLT03] Heckel, R., Lohmann, M., and Thöne, S. “Towards a UML profile for service-oriented architectures”. In: *Proceedings of Workshop on Model Driven Architecture: Foundations and Applications - MDAFA’03*. 2003.

[JZM07] Jiang, K., Zhang, L., and Miyake, S. “OCL4X: An Action Semantics Language for UML Model Execution”. In: *Proceeding COMPSAC’ 07*. IEEE, 2007, pp. 633–636.

[JK06a] Jouault, F. and Kurtev, I. “On the architectural alignment of ATL and QVT”. In: *Proceedings of the 2006 ACM Symposium on Applied Computing, SAC’06*. ACM, 2006, pp. 1188–1195.

[JK06b] Jouault, F. and Kurtev, I. “Transforming models with ATL”. In: *Proceeding of the 2005 International Conference on Satellite Events at the MoDELS*. Springer, 2006, pp. 128–138.

[KBC03] Kalnins, A., Barzdins, J., and Celms, E. “Model Transformation Language MOLA”. In: *Proceeding of the 2005 European Conference on Model Driven Architecture: Foundations and Applications*. Springer, 2003, pp. 62–76.

[Kap+09] Kapitsaki, G., Kateros, D., rezerakos, G., and Venieris, I. “Model-driven development of composite context-aware web applications”. In: *Information and Software Technology* Vol 51 issue 8 (2009), pp. 1244–1260.

[KW07] Kindler, E. and Wagner, R. *Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios*. Tech. rep. tr-ri-07-284. Software Engineering Group, Department of Computer Science, University of Paderborn, 2007.

[Lab13] LabVIEW. *System design software NI LabVIEW*. www.ni.com/labview/d/. 2013.

[LG08] Lara, J. and Guerra, E. “Pattern-Based Model-to-Model Transformation”. In: *Proceedings of ICGT’08*. Springer, 2008, pp. 426–441.

[Lia08] Liang, D. *Compiler for the Object Constraint Language (OCL)*. Master Thesis. Freiberg University of Mining and Technology. 2008.

[LS10] Liang, D. and Steinbach, B. “A new General Approach to Model Event Handling”. In: *The Fifth International Conference on Software Engineering Advances - ICSEA 2010*. 2010, pp. 14–19.

[LS11] Liang, D. and Steinbach, B. “Compact and Efficient Modeling of GUI, Events and Behavior Using UML and Extended OCL”. In: *International Journal on Advances in Software* Vol 4 nr 1&2 (2011). IARIA, pp. 100–116.

[Lóp+07] López-Sanz, M., Acuña, C., Cuesta, C., and Marcos, E. "Modelling of Service-Oriented Architectures with UML". In: *Proceedings of the 6th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2007).* Vol. 194. 2007, pp. 23–37.

[Rum02] Rumpe, B. "«Java»OCL Based on New Presentation of the OCL-Syntax". In: *Proceeding of Object Modeling with the OCL, The Rationale behind the Object Constraint Language.* Ed. by Clark, T. and Warmer, J. Springer, 2002, pp. 189–212.

[Sch94] Schürr, A. "Specification of Graph Translators with Triple Graph Grammars". In: *Proceedings of the 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science, WG'94.* Springer, 1994.

[Sel04] Selic, B. "On the Semantic Foundations of Standard UML 2.0". In: *Formal Methods for the Design of Real-Time Systems (SFM-RT 2004).* Vol. 3185 of LNCS. Springer, 2004, pp. 181–199.

[SP03] Silva, P. and Paton, N. "User Interface Modeling in UMLi". In: *Software* Vol 20 issue 4 (2003). IEEE, pp. 62–69.

[SP01] Silva, P. and Paton, N. "User Interface Modeling with UML". In: *Information Modeling and Knowledge Bases XII.* Ed. by Jaakkola, H., Kangassalo, H., and Kawaguchi, E. IOS Press, 2001, pp. 203–217.

[Ste12] Steinbach, B. *Lecture note of the course Software Technology and Software Project.* Freiberg University of Mining and Technology. 2012.

[SBL04] Steinbach, B., Besser, O., and Liang, D. *The UML Test Front-End Project.* http://www.informatik.tu-freiberg.de/index.php?option=com_content&task=view&id=53&limit=1&limitstart=5#UTEFEI. 2004.

[Ste+02] Steinbach, B., Dorotska, C., Rudolf, G., Bittner, S., and Liang, D. *The Generation of Test Cases Project.* http://www.informatik.tu-freiberg.de/index.php?option=com_content&task=view&id=53&limit=1&limitstart=4#GETECAI. 2002.

[SFB05] Steinbach, B., Fröhlich, D., and Beierlein, T. "Hardware/Software Codesign of Reconfigurable Architecture Using UML". In: *UML for SOC Design.* Ed. by Grant, M. and Müller, W. Springer, 2005, pp. 89–117.

[Suj+11] Sujeeth, A., Lee, H., Brown, K., Chafi, H., Wu, M., Atreya, A., Olukotun, K, Rompf, T., et al. "OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning". In: *Proceedings of the 28th Intl. Conference on Machine Learning.* 2011.

[Vit08] Vitzthum, A. "Entwicklungsunterstützung für interaktive 3D-Anwendungen Ein modellgetriebener Ansatz". PhD thesis. http://edoc.ub.uni-muenchen.de/9459/1/Vitzthum_Arnd.pdf, 2008.

[Wik13] Wikipedia. *LL Parser.* http://en.wikipedia.org/wiki/LL_parser#General_case. 2013.

[WJ04] Witthawaskul, W. and Johnson, R. *An Object Oriented Model Transformer Framework based on Stereotypes.* http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.123.1587&rep=rep1&type=pdf. 2004.