Dipartimento di Informatica, Bioingegneria,
Robotica ed Ingegneria dei Sistemi

MeDMoT: a Method for Developing Model to Text Transformations

by

Alessandro Tiso

Theses Series **DIBRIS-TH-2014-02**

DIBRIS, Università di Genova

Via Opera Pia, 13 16145 Genova, Italy `http://www.dibris.unige.it/`

Università degli Studi di Genova

Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi

Dottorato di Ricerca in Informatica

Ph.D. Thesis in Computer Science

MeDMoT: a Method for Developing Model to Text Transformations

by

Alessandro Tiso

March, 2014

Dottorato di Ricerca in Informatica
Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi
Università degli Studi di Genova

DIBRIS, Univ. di Genova
Via Opera Pia, 13
I-16145 Genova, Italy
`http://www.dibris.unige.it/`

Ph.D. Thesis in Computer Science (S.S.D. INF/01)

Submitted by Alessandro Tiso
DIBRIS, Univ. di Genova
`alessandro.tiso@unige.it`

Date of submission: March 2014

Title: MeDMoT: a Method for Developing
Model to Text Transformations

Advisor: Gianna Reggio
Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi
Università di Genova
`gianna.reggio@unige.it`

Ext. Reviewers:

Alfonso Pierantonio
Università degli Studi dell' Aquila
`alfonso@di.univaq.it`

Marco Torchiano
Politecnico di Torino
`marco.torchiano@polito.it`

Abstract

Model driven methods are software development approaches defined in the last decade that are based on the concept of model. All these methods aim to raise the abstraction level from code to models: in practice, the code is generated from the models by means of automatic transformations or the models are directly executed. Model driven methods are declined with different acronyms like MDA, MDD, MDE etc., there are some differences between the meaning of these acronyms, since there is not a general agreement about these meaning, we use the MD* acronym to denote all these acronyms. At the heart of all these techniques there are model transformations. They must be considered as primary artefacts when MD* techniques are applied. Like any other piece of software, model transformations must be designed, implemented and tested. These tasks are in general more difficult that the equivalent performed on traditional software, thus there is a need to define a method to support the model transformation development. The goal of this thesis is to define a general method for developing model transformations whose application is effective in real world applications, like the model driven generation of enterprise applications. In the context of this main goal, we have the following sub-goals: (1) study the diffusion, state of the practice, adoption and effectiveness of MD*; (2) define methods for the requirements definition, the design and the testing of model transformations; (3) select a set of integrated tools to support the development of model transformations; (4) evaluate the proposals of (2), (3) and (4) by means of the application to a non-toy case study. Having these goals in mind we studied by means of empirical techniques: the state of the practice of MD* in the Italian industry (industrial survey), the maturity of software modelling and MD*, the expectation and the achievements using modelling and MD* (industrial survey), the effectiveness of model driven in maintenance tasks, by means of an experiment (controlled experiment). We have defined a general method for developing model transformations named *MeDMoT* covering the requirement definition, design, implementation and testing phases. In particular we have: (1) defined a notation for presenting the design of model transformations; (2) selected appropriate model transformation languages and a technology environment with a set of integrated tools to support all the activity that must be performed in the implementation of model transformations; (3) defined methods to test model transformations; (4) applied *MeDMoT* to the case of the complete generation of Java desktop applications (except the GUI) using up to date technologies (like the Spring framework, JPA with Hibernate and Maven), from a specific UML design model. The developed transformation has led to the building of a tool for the generation of desktop applications that we call AutoMARS.

To my wife, who always supports me, and to the memory of my father.

Acknowledgements

I want to thank my advisor Prof. Gianna Reggio for the continuous support of my Ph.D study and research.

Besides my advisor, I want to thank Prof. Filippo Ricca for his encouragement and insightful comments and Prof. Egidio Astesiano for his words of encouragement.

I thank Maurizio Leotta, for the stimulating discussions and for the hours in which we were working together before deadlines.

Last but not the least, I want to thank my wife Laura and my children Leonardo and Alberto for their patience.

Table of Contents

Chapter 1

Introduction

Models are relevant in every engineering activity like for example structural engineering, electrical engineering and naval engineering. Models has been used so extensively by engineers, because they are a way to cope with the complexity of the reality. Indeed, models are abstraction of real systems, in which are highlighted the properties of interest for a given viewpoint.

In software engineering, systems are composed by programs (and data structures), thus they are key artifacts of the software development process. In this kind of process, models has been traditionally used in design or documentation. Programs composing software systems are manually written starting from design models or documentation models are built observing existing software systems. Since the code is manually written, this kind of models tend to be out-of-sync with the represented systems and hence they quickly become not very useful. Moreover, software practitioner feel the activity of maintain in sync code and models as an overhead that does not have a justification, especially after the first version of the system is built starting from design models [99]. A way to avoid this problem is use models as primary artifacts of the software development process, automating the translation from models to code or directly executing models. When models are put at the core of the software development process, then the process is called *model driven*.

There are different model driven techniques: Model Driven Engineering (MDE) [96], Model Driven Development (MDD) [87] and Model Driven Architecture (MDA) [82]. MDD is a development approach which focuses on the use of models as key artifacts of the software development cycle, and has as main goal to raise the level of abstraction at which software engineer create software. In this development approach models at high level of abstraction are transformed into models at lower level of abstraction until the models is executable using either code generation or model interpretation. MDE is a generalization of MDD, which embraces more Technological Spaces [83] and encompass other activities besides the development. Finally, MDA is a registered trademark of Object Management Group (OMG). It is an instantiation of MDD which

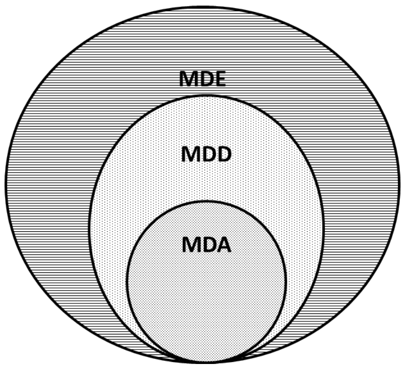


Figure 1.1: MDE, MDD and MDA Relations

uses a set of OMG specifications on which MDD may be built. In Fig. 1.1 is shown a diagram highlighting relations among MDE, MDD and MDA [42].

Besides those acronyms, there are others used with more or less the same meaning (e.g., MDSD[1], MDSE[2]), thus when the differences between these approaches are not relevant, taking inspiration from [114] we refer to them with the acronym MD*. Models are specified in some modelling language, which can be visual or textual, tailored to a specific domain (in this case, they are called Domain Specific Languages, DLSs) or general purposes (e.g. the Unified Modelling Language, UML). In all these cases, a modelling language to be useful for a MD* approach, must have a well defined syntax and semantic [99].

Starting from the early works on MDA, MDD and MDE (respectively in 2000 with the white paper [103], in 2003 with [87] and 2006 with [96]) a lot of work has been done in the MD* context. Actually, there are many specific model driven methods supported by commercial and open source tools (e.g. WebRatio[41], AndroMDA[3] and BridgePoint[4]).

First of all, we wanted to to investigate about the dissemination and the effectiveness of modelling and model driven techniques during software development and model driven usage. We approach this task using techniques proposed by the Empirical Software Engineering. We conceived and designed a survey with the goals of understanding: the actual relevance of software modelling and model driven approaches in the Italian industry, the way software modelling and model driven are applied, and the motivations either leading to the adoption (expected benefits) or preventing it (experienced or perceived problems). Moreover, we have conducted an experimental assessment of maintenance tasks with model driven techniques.

Model transformations are the heart and the soul of MD* approaches, and must be considered as

[1]MDSD stand for Model Driven Software Development

[2]Model Driven Software Engineering

[3]`http://www.andromda.org/index.html`

[4]`http://www.mentor.com/products/sm/model_development/bridgepoint/`

primary artifacts when MD* techniques are applied [100].

Moreover, model to text transformations are used also when MD* approaches are not completely embraced, like for example in the production of input for simulation tools or performance analysis [44]. At the time we are writing there are several languages and tool to support model transformations, but there is a lack of methods to develop model transformations (covering all the development phases), since this topic is not extensively considered by the literature.

We think that the same software engineering principles which are applied in traditional software development to make the software production more systematic and quantifiable, and hence more effective, should be applied when producing a model transformation..

This means that developing model transformations, we must take care of all the development phases such as: requirements definition, design, implementation and testing; and that there is a need of a (possibly integrated) set of methods/techniques/notations to support the developers during all those phases, and thus covering the whole transformation development process.

We can distinguish two main kind of model transformation [61]: the model to model transformation approach in which the target of the transformation is one or more models, and the model to text transformation approach in which the target of the transformation is text.

In the thesis we concentrate only on the model to text transformations. Model to text transformations not only are used in the last steps of a complete MD* software development process to produce the files containing code and the configuration files defining a modern software system, but may be the way to allow user of any kind to perform tasks of different nature working with visual models instead of the scarcely readable text required by software tools (e.g., the structure of a database may be designed using a UML class diagram instead of a bunch of XML based configuration files for a DBMS). Moreover, model to text transformations may be used also to generate automatically textual documentations.

We have discarded the idea of working only with model to model transformations, converting model to text transformations into model to model transformations, by defining additional metamodels for the naturally textual targets to transform them into models. The negative aspects of converting texts into models from our point of view are:

- the overhead due to produce the additional metamodels; in many cases the definition of these metamodels is neither very natural nor they can be found already available (think for example of a metamodel for Hibernate); in other cases the transformation target is made by text of different nature (e.g., Java code, Hibernate configuration files, HTML code, thus the required metamodel becomes also quite large and difficult to manage);
- an additional transformation from model to text has to be defined to get the needed textual artifacts, and this is another overhead;
- the conception and implementation of the transformation should be made at the metamodel

level for what concerns the target, whereas the available knowhow is usually in terms of the textual version (compare speaking of the metaclass “Class” for Java versus speaking of a class declaration in Java).

On the other side we have not been able to find real advantages of transforming the textual targets into models, except the fact that conceptually it is better to work in a homogeneous world where everything is a model.

Currently, to the best of our knowledge, in the literature most of the works about model transformation development deal with the implementation phase, and only few works cover the whole development process [71, 70]. Furthermore, almost all proposals deal with the development of model to model transformations. Moreover, each phase of the model transformation development presents problems difficult to solve, and for them there are not yet well defined and shared solutions, especially in the case of model to text transformations. Indeed, developing model to text transformations is a problem more complex than developing software due to the complexity of input and output.

Proposals of methods for capturing and representing model transformation requirements are very rare in literature. Again to the best of our knowledge, we can cite only [71, 70], and [67] as works that attempt to describe methods to capture model transformation requirements, although both of them deal with model to model transformations.

Again, model transformation design methods found in literature generally deal only with model to model transformation, although there is not a standard approach on how to perform the transformation design and the notation that has to be used.

Implementing model transformations requires to choose among a plethora of model transformation languages [74] and supporting tools which are not always interoperable and integrated.

Model transformation testing is a challenging problem [52]. Moreover, this is a young field of research and the proposed solutions deal with model to model transformation being [59] an exception since it considers model to text transformations.

Thus, there are no available proposals for a method for developing model to text transformation.

For these reasons, we have defined a general method for developing model to text transformations that we present in this thesis. This method covers all phases of the development of model to text transformations, providing specific techniques and notations for each phase. It supports the development of any kind of model to text transformations, not only from model to code, but also from models to various textual artifacts.

Moreover, we have conceived this method to be quite lightweight and applicable to the development of transformation of reasonable size. Since in our case model transformation targets are very heterogeneous, it is difficult to propose a method which is effective in all cases and that details precisely all the required activities , especially regarding the testing phase. A model transformation method which considers a specific type of target should take advantage of the available techniques, methods and tools for that target (e.g., if the target are Java applications, then well-established architectural styles and standard testing techniques should be considered while developing a transformation from UML models into Java). The proposed method, although general, do not lose this advantage.

The method proposed considers model to text transformations, and prescribes how to develop a transformation guiding to:

- capture and specify the transformation requirements;
- design the transformation;
- implement the transformation;
- test the transformation.

For each phase covered by our method we give our contribution.

Writing models suitable to be transformed is not so easy [99]. Indeed, models can be written in a quite freely way, producing also incorrect models (models not conform to the specification of the modelling language) (e.g. using the UML is possible to write this kind of models) or incomplete models. Furthermore, developing a transformation from quite loose models, even if they are correct, may be quite hard having to cope with a large set of cases. Thus, we have defined an approach for modelling using UML specific items for specific aims, which we have called Precise Modelling Method [46, 47]. Precise Modelling Method prescribes that in any modelling task we should determine a class of UML models, or more precisely a class of models written using a specific UML profile, whose form is precisely defined, by means of a (conceptual) metamodel and by a set of well-formdeness rules.

Our method for developing transformations requires that the source of the transformation must be defined following the Precise Modelling Method approach.

We called our method for developing model to text transformation *MeDMoT* (**Me**thod for **D**eveloping **Mo**del to **T**ext transformations).

We validate our method applying it to three case studies of different size:

- a toy example (that will be used through the thesis as the running example), a transformation from UML models of relational data bases to SQL DDL (U2SQL),
- a medium size case, a transformation from UML models of ontologies to OWL (U2OWL),
- a large size case, a transformation from UML models of desktop applications to Java desktop applications (U2Java).

The implementation of the U2Java transformation has lead us to develop a tool, that we have called AutoMARS. AutoMARS is a model driven application generator that starting from a UML model representing a detailed design of an application can generate complete (excluding the GUI) Java desktop applications built using up-to-date technologies

1.1 Outline of the Thesis

The thesis is structured as follows:

Chapter 2 describes the basic concepts used in thesis and the state of the art. It introduces MDD, MDA, MDE and model transformations. Moreover, it gives a classification of model transformation approaches and languages.

Chapter 3 presents the survey used to study the state-of-the-practice of modelling and model driven in the Italian industry and relative findings. Furthermore, it describes the experiment done to assess the effectiveness of model driven approaches in maintenance tasks.

Chapter 4 describes (Precise Modelling Method) our approach for defining methods for using the UML for modelling specific items for specific aims. Moreover, in this chapter we show how to apply Precise Modelling Method to three cases: MRelational a simple modelling method for relational data bases, MOntology a method to use UML to model ontologies, and MJavaD a method to model design specification of Java desktop applications.

Chapter 5 describes *MeDMoT*, our method for developing model to text transformations. In this chapter we introduce: (1) a method to capture transformation requirements, (2) guidelines and notation to design a transformation, (3) guidelines on how to test a transformation and finally (4) guidelines on how to test a transformation.

Chapter 6 presents the two case studies to which our method is applied: U2OWL and U2Java. Moreover, it describes AutoMARS a tool that is a model driven application generator that starting from a UML model representing a detailed design of an application can generate complete (excluding the GUI) Java desktop applications built using up-to-date technologies.

Chapter 7 gives conclusions and presents ideas for future works.

Chapter 2

Model Driven Techniques and Model Transformations: Basic Concepts

In this chapter we give an overview of the concepts used in the thesis. We give a brief description of model driven techniques, such as model driven development, model driven architecture and model driven engineering. Moreover, we briefly describe model transformation approaches and model transformation languages based on these approaches.

The structure of the chapter is as follows: Sect. 2.1 gives an introduction to model driven development and model driven architecture.Moreover, in the same section we give some definitions about models. Sect. 2.2 gives a description of model driven engineering.In Sect. 2.3 we describe model transformations and model transformation languages, giving also a classification of the several model transformation approaches. Sect. 2.4 give a conclusion of the chapter.

2.1 Model Driven Development

Models are relevant in every engineering activity like for example structural engineering, electrical engineering and naval engineering. Also in software engineering models are often used for many purposes like, for example, design and documentation, but they are not considered the core artifacts produced during software development. Models used in these ways tend to be out of sync with the code, and for this reason, their importance is not properly considered.

In the last fifteen years software development approaches have been developed that put models at the core of software development activities. One of these approaches is Model Driven Development (MDD), which focuses on the use of models as key artifacts of the software development cycle.

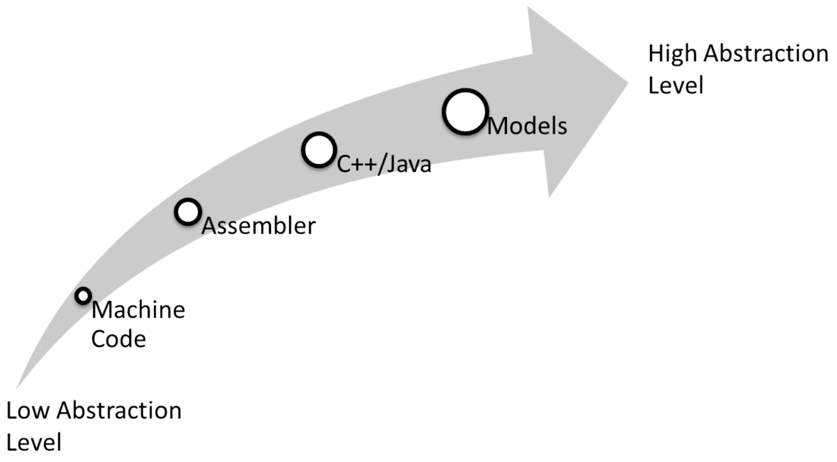


Figure 2.1: Abstraction Level

The main goal of MDD is to raise the level of abstraction at which software engineer create software reducing the complexity of the produced software artifacts and the effort needed to build them [73]. Indeed, by raising the level of abstraction, developers can write software specifications closer to the problem domains, more independent from platform details (thus more insensitive to platform changes) and can cope with the increasing complexity of the software systems to be produced [99, 73].

Starting from assembly languages to four generation languages, the abstraction lift was one of the key techniques to manage complexity and improve productivity, see Fig. 2.1.

Other motivations (not disjoint from the main goal) for MDD is to improve short-term and long-term productivity [48]. The former is achieved increasing the number of functionality delivered by each artifact (models in MDD). To assure an increase of productivity, in MDD models are transformed into programs or directly executed, leveraging a high level of automation. Thus, transforming models is one of the most important activities that must be performed in MDD (model transformations are described in more detail in Sect. 2.3). The latter depends on how the artifacts produced are sensible to changes (change of deployment platforms, for example). In MDD this is achieved using model transformations for targeting different kind of platforms.

To use models as primary artifacts, MDD needs modelling languages, visual and/or textual, that have semantics and syntax definition as precise as possible. Those languages can be general purposes or tailored for a specific domain (in this case they are called Domain Specific Languages, DSLs).

Moreover, to support a complete life-cycle based on the use of models as key artifacts, MDD

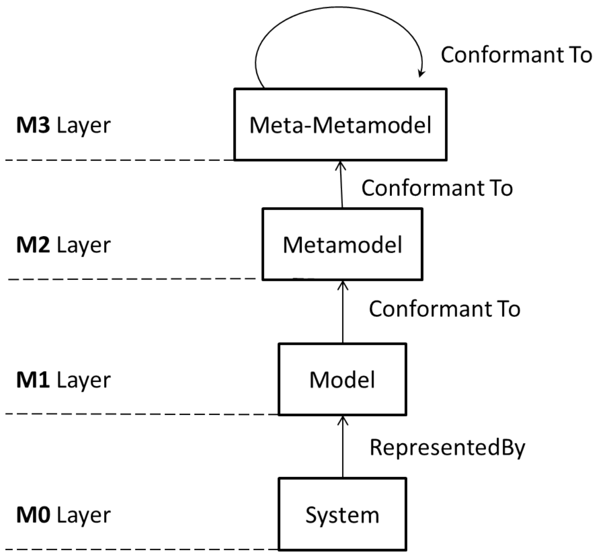


Figure 2.2: Four Layers Metamodelling architecture

needs a modelling infrastructure.

Before we can describe a modelling infrastructure, may be useful give some definitions about models. In literature there several definitions of what a model is, here we report some of them. A model of some system can be defined as: “a reduced representation of that system that highlights the properties of interest for a given viewpoint” [99], or more mathematically, “a set of statement about a system under study (SUS)” [98], or even “model is a representation of a part of the function, structure and/or behaviour of a system” [86]. A modelling language, that is a language with which we can write models, is defined by syntax and semantics which can be defined building a model of the modelling language, the so called metamodel. A way to specify a metamodel is using a language, which can be specified using a meta-metamodel. The meta-metamodel is specified in term of itself.

Now we can describe one of the most used modelling infrastructure that is the one called four-layer metamodelling architecture Fig. 2.2. These modelling infrastructure has been used by OMG placing some standard at each level. We describe these standard in the next paragraph.

Layer M0 this is the concrete level. At this level we have what is to be modelled.

Layer M1 this is the model level. At this level there are all the models each one instance of some metamodel in the M2 layer.

Layer M2 this is the metamodel level. At this level there are all the metamodels each one conforms to the meta-metamodel at level M3. Each metamodel describe the abstract syntax of some of the models in the M1 layer.

Layer M3 this is the meta-metamodel level. A meta-metamodel is used to specify the modelling language used to express metamodels in the M2 layer. The language used to specify the meta-metamodel language is reflexively specified, thus no other higher layer is needed.

As in other software development approaches, tool support is essential. Since in MDD models are primary artifacts, tools must be able to build and manage models at least like tools help to build and manage code in traditional code centric software development methods. Thankfully, today support tools are mature enough to support all the MDD activities.

Model Driven Architecture (MDA) is a particular realization of MDD issued by OMG [1], that is described in Sect. 2.1.1. Moreover, MDA is a concrete example of MDD, thereby describing MDA we better explain MDD.

2.1.1 Model Driven Architecture

Since MDA is an instantiation of MDD, also in MDA the development of an application or a system can be viewed as a set of model transformations from models at high level of abstraction to models at low level of abstraction.

OMG provides a set of detailed specifications, on which the MDD can be built. Among these specification, the following are those categorized as OMG modelling specifications, that provides a foundation for MDA:

The Unified Modelling Language (UML) [25], is graphical language for modelling systems. It has various diagrams enabling modelling of various aspects of a system, from structural to behavioural.

XML Metadata Interchange (XMI) [21], define a mechanism for interchange among tools, repositories and middleware. It provides an XML serialization mechanism for UML and MOF models.

Meta Object Facility (MOF) [20], provides constructs for modelling and interchange used in MDA. It has the ability of define modelling languages.

Common Warehouse Metamodel (CWM) [19], is a metamodel that define a standard interface between warehouse tools, warehouse platforms and warehouse metadata repositories.

OMG place some of these standards in the four-layer metamodelling architecture (see Fig. 2.2) [98, 20].

[1]The Object Management Group (OMG) is an international, technology standards consortium, see `http://www.omg.org/`

The only meta-metamodel is the MOF and is placed at the M3 layer; at M2 layer there are metamodels as the UML and the CWM; at M1 layer there are all the models written using the UML language and finally at the layer M0 there are concrete instances of models.

Therefore, in MDA models are written using the UML, which is specified by means of the UML metamodel, that in turn is specified by the MOF, which is defined in term of itself.

Having that UML is a general purpose language, it may happen that UML is not perfectly suitable for particular domains. OMG has provided two ways to extend UML [66]: the former is based on the definition of a new language using the UML metamodel, the latter is based on the profile mechanism. The profile approach is a set of extension mechanisms (stereotypes, tagged values and constraints) that allow to customize UML restricting the number of UML elements, adding some syntactic sugar to them or modify the semantic of UML elements. UML profiles have an important role in MDA because they simplify the description of transformation rules between models. OMG itself has defined some profiles, like for example:

- UML Profile for MARTE: Modeling and Analysis of Real-time Embedded Systems (MARTE) [32];
- UML profile for Enterpise Application Integration (EAI) [31];
- UML profile for Modelling QoS and Fault Tolerance Characteristics and Mechanisms (QFTP) [33];
- UML profile for Schedulability, Performance and Time (SPTP) [34];
- UML profile for System on a Chip (SoC) [35];
- UML Testing Profile (UTP) [36];

It is important to point out that in MDA the concept of model abstraction is relative to the concept of *platform* and *platform independence*. MDA define the concept of *platform* as all the details that are irrelevant to the functionality of the software component. Thus, in MDA the meaning of the term *platform* is strongly context sensitive and is relative to a particular point of view [58]. Models at high level of abstraction are models with an high level of independence from the *platform*; conversely model at low level of abstraction are models with a low level of independence from the *platform*.

To support the development process in MDA are defined a set of views, or viewpoint models, that are models representing the system at different levels of abstraction.

Each view focus on a particular concern of the system, these views are [111]:

Computation Independent Model (CIM) this model presents what the system is expected to do, but all the information technology related are hidden. It is used to model the system requirement, and should be traceable with the other models of the system.

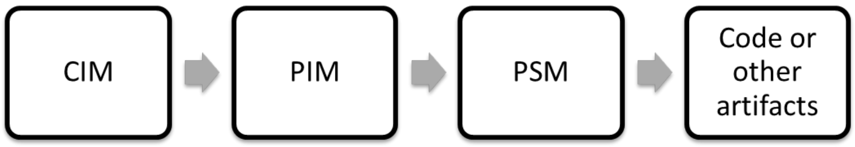


Figure 2.3: MDA Basic Process

Platform Independent Model (PIM) this model does not contain any of the details specific of the target *platform*, so it can be mapped to one or more *platforms*. It shows a certain degree of *platform independence*.

Platform Specific Model (PSM) this model contains all the details needed to specify how the system uses a particular type of *platform*. It combines the PIM with the details that specify how the system modelled by the PIM uses a particular type of *platform*.

MDA defines an application building process that can be summarized as follow [111, 102]: first a CIM must be built to capture the requirements of the application, then a PIM must be developed adding architectural knowledge to the CIM, after that PIM is transformed into a PSM (adding to the PIM information relative to a specific platform). It is important to note that the same PIM can be transformed into more than one PSM, depending on the chosen target platform. Finally, the PSM can be then transformed into code or other artifacts (see Fig. 2.3).

MDA Model Transformations

Model transformations play a basic role in MDA. In the MDA reference documentation [113] the important case of the transformation between the PIM and the PSM is discussed.

The process of transforming a model in MDA is defined as: "the process of converting one model to another model of the same system" and is based on the concept of *mapping*. There are two types of *mapping*, the *model type mapping* and the *model instance mapping*. The former is a mapping between types of the PIM language (the types specified in the language used to build the PIM) to types of the PSM language. The latter is a mapping between PIM elements and *marks*, which represents concepts in the PSM to indicate how PIM elements are to be transformed. Thus, in MDA there are two ways to transform a PIM into a PSM, one for each type of mapping [113].

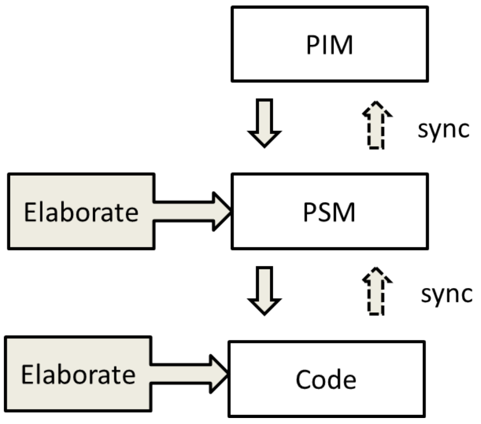


Figure 2.4: MDA Elaborationist intepretation

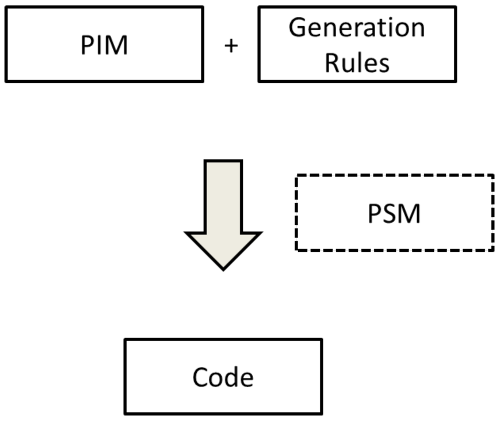


Figure 2.5: MDA Translationsist intepretation

MDA Interpretations

There are two interpretation of MDA [85]: the *elaborationist* Fig. 2.4 approach and the *translationsist* approach Fig. 2.5.

In the former approach, a first sketch of a PIM is built, then it is transformed into a PSM that the developer can elaborate before the following transformation into the code (or other artefacts suitable for the chosen platform). The code itself can be modified by the developer and the changes made can be reflected in the upper level (PSM) using another transformation. In the same way the changes made in the PSM can be reflected in the PIM. The process of transforming models, apply some modifications to the target of the transformation and then transforming back the result into the model, so it can reflect the changes made in the target of the transformation is called *round-trip engineering*. The developer can apply one or more round-trip cycles until he is satisfied of the result, maintaining the PIM, PSM and code in sync.

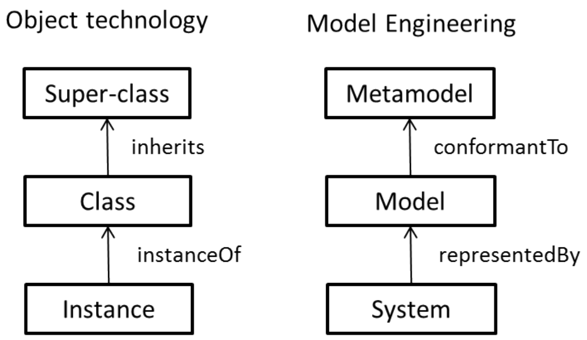


Figure 2.6: Basic Notions in Object Technology and Model Engineering

In the latter approach the PIM is directly transformed into the code using a more sophisticated transformation (w.r.t. the transformations used the *elaborationist* approach) that contains rules that describe how the PIM must be transformed into the code. Differently from the former approach, the target of the transformation is not further elaborated.

2.2 Model Driven Engineering

It is difficult to define precisely what is Model Driven Engineering (MDE), but we can think to MDE as a generalization of MDD. In the following we want underline what are the differences between MDD and MDE.

Kent in his paper [79] tries to set out a framework for model driven engineering that proposes an organization of modelling space, explaining why process and architecture are tightly coupled and discuss about the importance and nature of tools. Moreover, he identifies the need for defining family of languages and transformations. One way to organize the modelling space is to categorize the perspectives that model is intended to take, so we can thought at the criteria that can be used to choose the perspectives as different dimension in an n-dimensional space. MDA defines only one dimension in the modelling space that reflects the degree of abstractness and concreteness of models (PIM and PSM are models of the same system at different level of abstraction). Kent, identifies three modelling dimensions: one dimension of concern is the *subject* area, that distinguishes models on the base of the subject are they belong to; another dimension of concern reported is the *aspect* (for example concurrency of control and distribution) and a third dimension is concerned with managerial and societal aspect such as authorship, version, location and stakeholder. He states that when building a model, one must identify at what intersection of the dimensions the model should be placed.

Bézivin in [54] tries to set out the basic principles of MDE. He found these principles in the concepts of model and system (described in Sect. 2.1) and the two relations of conformance and representation (see the right side of Fig. 2.6). Moreover, he compares the basic principles in object technology "everything is an object" with the basic principle of model engineering "everything is a model", as it is summarized in Fig. 2.6. What was important in object technology is that an object can be an instance of a class, and a class can inherits from another class, so we can call the two corresponding basic relation *instanceOf* and *inherits*. For the model engineering is important that each particular aspect of a system can be captured by a model and each model is written in the language defined by its metamodel, so we can call the two corresponding relation *representedBy* and *conformantTo*. The two principles should be viewed as complementary approaches.

Furthermore, one can note that the classical four-level architecture of Fig. 2.2 is similar to the organization of programming languages. Indeed, a self-representation of an Extended Backus-Naur Form (EBNF) notation allows defining infinity of well-formed grammar. Given a specific grammar (for example the grammar of the Java language), allows defining of infinity of syntactical correct Java programs, and in turn a single Java program is a symbolic representation of the infinity of its possible executions.

Technical spaces

Recognizing the similarities between the four-layer meta-model architecture and other architectures, Kurtev et al. in [83] and then Bézivin et al. in [55] define the concept of Technological Spaces (in short TSs). Kurtev et al. [83] give a definition of TS as "A technological space is a working context with a set of associated concepts, body of knowledge, tools, required skills and opportunities". Each technical space can be seen as a three level organization composed by one meta-metamodel, one (or more) metamodel/s conforms to the meta-metamodel and models conform to one of the metamodels belonging to the TS.

For example in the MDA TS the meta-metamodel is the MOF and one of the metamodels may be the UML metamodel. There are other TSs, they also organized in a three level organization, like for example: (1) the XML TS in which a XML document must be written in a syntax constrained by well-formedness (XML grammar) and validity (DTD or XML-schema) rules; (2) the abstract syntax TS in which a program written in a given programming language whose syntax is specified in a grammar.

Fig. 2.7 shows a representation of the TS concept (the image in the upper left corner) and two instantiation of this concept representing the MDA TS (the image in the upper right corner) and Abstract Syntax TS (the image in the lower left corner).

Moreover, TSs are not isolated, instead there are conceptual and operational bridges among them, so is possible to use more than one TSs in synergy to solve a specific problem. These bridges among TSs boundaries may be one or bi-directional and are called *projectors*. There are two kinds of projectors accordingly with the direction: *injectors* and *extractors*. Obviously, the

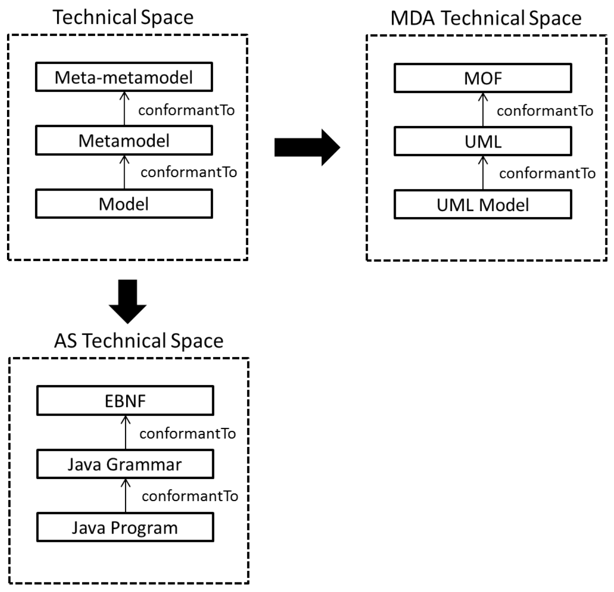


Figure 2.7: Technical spaces

decision to use or not projectors depend on cost and effort considerations.

2.3 Model Transformations

Model transformations are at the core of model driven engineering enabling the capabilities to create, merge, refine or refactor models. They, and their maturity, are needed to model driven to become a reality. Moreover, model transformations must be described and executed in an effective way, thus a plethora of model transformation languages and supporting tools were developed and maintained.

In this section, we briefly describe model transformations approaches and model transformation languages based on these approaches.

There are a number of general definition of MT such as the one given from Kleppe et al. [81]: ”*A transformation is the automatic generation of a target model from a source model, according to a transformation definition.*”. The same definition is then generalized in [89] considering the possibilities of multiple source models and/or multiple target models (see Fig. 2.8).

A *transformation definition* is a description of how the source model is transformed into the target model. It can be written in a general purpose language (e.g., Java) or in a *transformation language*. Each *transformation language* is based on a certain *model transformation paradigm* (also called *model transformation approach*), that is the design principle on which the model

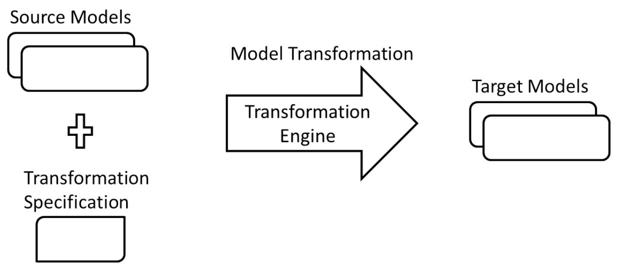


Figure 2.8: Component of Model to Model Transformation

transformation language is built [57]. Obviously, the transformation definition must be executed by a transformation engine (e.g., the Java virtual machine, if the transformation language is Java), which apply the transformation definition on the source model/s to produce the target model/s.

Since, model driven approaches has been studied for several years, there are a lot of MT languages designed following several MT paradigms. Moreover, MTs have been used in different scenarios.

Model transformations and MT languages has been characterized and classified in several works. In the following we briefly report on the classification of MT languages described in [62, 89], and on the classification of problems that can be solved using MT, described in [57] and in [74].

2.3.1 Model Transformation Classification

Model transformations can be used to solve different kinds of problems, each of which is an instance of a different usage scenario.

When a MT is used to change the level of abstraction, it is a *vertical transformation*. Obviously, the level of abstraction can be increased or decreased by the MT. In former case it is a *refinement transformation* and in the latter case it is an *abstraction transformation*. Conversely, an *horizontal transformation* changes the representation of the model but does not change the level of abstraction of the model.

When the metamodels of the source and the target of the MT are not the same, the MT is called *exogenous transformation*. Conversely, when the source and target metamodels are the same, the MT is called *endogenous transformation*.

Model transformation can be classified as follows:

We can distinguish two main MT approaches: (1) the *model-to-model* transformation approach in which the target of the transformation is one or more models and (2) the *model-to-text* transformation approach in which the target of the transformation is text. In the latter case the model transformation definition given above may be modified as follows: "*A transformation is the au-*

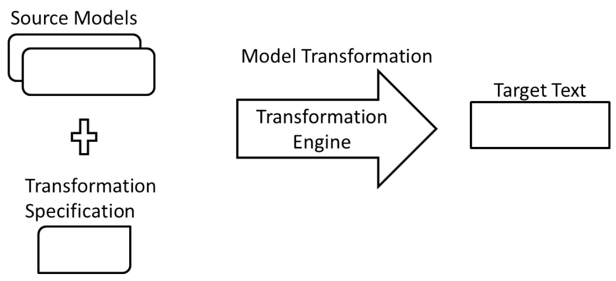


Figure 2.9: Component of Model to Text Transformation

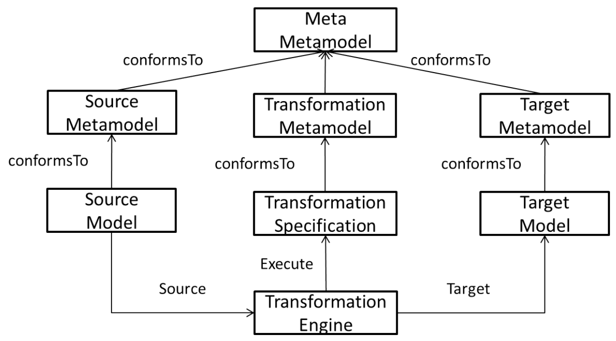


Figure 2.10: Model to Model Transformation in the context of a Metamodelling infrastructure

tomatic generation of a target text from one or more source model, according to a transformation definition" (see Fig. 2.9).

MT can be viewed in the context of a modelling infrastructure such that described in Sect. 2.1 and showed in Fig. 2.2. In this case, we may have a MT specification that is a model itself, this model must be conform to a metamodel which, in turn, is conform to the meta-metamodel of the TS (in each TS there is only one meta-metamodel). The meta-metamodel is the same also for the metamodels of the source and the target models (which may not be the same). This situation can be shown as in Fig. 2.10.

Also the model to text transformation can be viewed in the context of a modelling infrastructure. Obviously, in this case there is not a target metamodel (see Fig. 2.11).

In general model to text transformations can be viewed as a special case of model to model transformations, if target/s metamodel/s is/are provided. However, for simplicity if the target of the transformation is some kind of text, it is directly generated by the transformation.

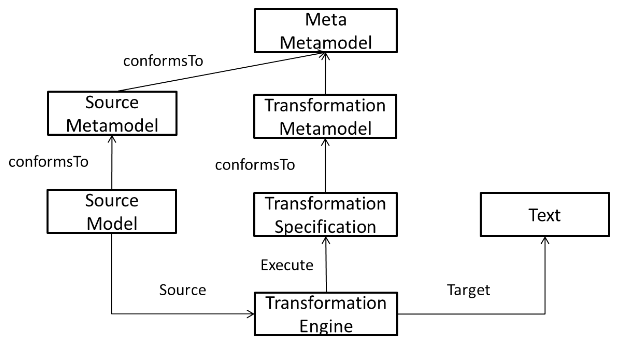


Figure 2.11: Model to Text Transformation in the context of a Metamodelling infrastructure

Model to Text Approaches

In this category we can distinguish between *visitors based*[2] and *template based* approaches.

Visitor based. A model transformation language based on this approach, must provide some visitor mechanism to traverse the internal representation of the source model, so as to write text on a text stream.

Template based. This one is the most used from available tools. A template consists in some target texts containing place-holders which, during the execution, are replaced by data extracted from the source model/s. Languages based on the template approach are: *Acceleo* [1] that is a pragmatic implementation of MOF model to text language (MTL) [22] language specification by OMG; *XPand* [39] that is a Domain Specific Language (DSL) for code generation; Java Emitter Template (JET) *Jet* [15] is an engine that allows text generation starting from EMF [10] models; *velocity* that is very simple Java-based template engine;

Model to Model Approaches

Operational/Imperative. Languages based on this approach have constructs similar to those of general purpose languages. They have constructs to sequentially control the transformation flow and like general purpose languages may express the transformation as a sequence of actions. Examples of languages based on this approach are: *QVT Operational* [28] a partial implementation of the Query/View/Transform Operational mapping language specification by OMG [17] and *Kermeta* [16].

[2]it is a pattern that can be used to traverse graph, see `http://en.wikipedia.org/wiki/Visitor_pattern`

Declarative/Relational. Declarative/Relational languages are based on the mathematical concepts of relations. Using declarative/relational languages a developer must describe the relationships between source and target metamodels, focusing on what should be mapped by the transformation . Moreover, languages based on this approach do not offer an explicit control of flow. An Example of language based on this approach is *QVT Relations* a partial implementation of Query/View/Transform Relations language specification [17].

Hybrid. Hybrid languages have both imperative and declarative language constructs. Examples of languages based on this approach are: *Query/View/Transform QVT* it has three separate components *QVT Operational*, *QVT Relations* and *QVT Core*, it is; ATL [77, 75, 76, 3] that is a QVT like language [78] in which transformation rules can be described in declarative way, in imperative way or using both ways.

Graph Transformation. This approach has its basis on theoretical works on graphs, in particular with Triple Graph Grammars [97] that are a way to describe graph transformations where each transformation rule is specified by three graphs: the source graph, the target graph and the correspondence graph which describe the mapping between element of the source graph and element of the target graph. An example of language based on this approach is the *VIsual Automated model TRAnsformations VIATRA* language [38]

Direct Manipulation. Using this approach a general purposed languages, such as Java or C++, and specific libraries can read and write model data, thus implementing a model transformations.

There are many other model transformation languages not reported here, for a list a description of these languages see [62, 74, 57] although, some of those reported in these works may be outdated.

2.4 Conclusion

In this chapter we gave a brief description of model driven approaches like MDD, MDA and MDE. Moreover, we gave some definition about models (models, metamodels and meta-metamodels) and we have introduced the four layer metamodelling architecture. Furthermore, we gave some notions about model transformations and model transformation languages, introducing also a classification of model transformation approaches.

Chapter 3

Empirical Studies on the Dissemination and Effectiveness of Model Driven Techniques

Among the objectives of the thesis there is to investigate about the dissemination and the effectiveness of modelling and model driven techniques during software development and model driven usage. We have approached this task using techniques proposed by the Empirical Software Engineering (ESE).

A definition of ESE could be: "Empirical software engineering is a field of research that emphasizes the use of empirical studies of all kinds to accumulate knowledge. Methods used include experiments, variety of case studies, surveys, and statistical analyses" [1].

There are several types of studies that apply the empirical approach. They can be classified dividing into two sets of empirical studies: *qualitative* and *quantitative*. The *qualitative* studies have as goal to provide knowledge of an organization and/or a problem and its solution. They have an exploratory motivation and often they are applied to a small amount of unstructured data using non statistical methods for the data analysis. On the contrary *quantitative* empirical studies have as goal to generalize the results of a case study to the population of interest. They have a confirmatory motivation and often they are applied to a large amount of structured data using statistical methods.

We are interested in particular at quantitative studies. Example of quantitative studies are:

Experiment it aims to identify a precise relationship between variables chosen by means of a designed lab situation. In this kind of studies are used quantitative and analytical techniques.

[1] `http://en.wikipedia.org/wiki/Experimental_software_engineering`

Survey it is a snapshot at a particular point in time. In this kind of study relationship inferences are made using quantitative and analytical techniques.

Case study it describes existing relationships, usually, within a single organization.

To study the diffusion of modelling and model driven techniques we conducted a survey presented in Sect. 3.1, instead to study the effectiveness of model driven techniques we have done a controlled experiment to investigate the impact of following a model driven approach (that is presented in Sect. 3.2), during software maintenance and evolution activities.

3.1 Diffusion of Modelling and Model Driven Techniques: a Survey

Since in this study we are not interested in a specific model driven technique, e.g., MDA (see Sect. 2.1.1), MDD (see Sect. 2.1), and MDE (see Sect. 2.2) and inspired by [114], we will address them with the abbreviation MD*.

The survey has been conducted through the following six steps [80]: (1) setting the objectives or goals, (2) transforming the goals into research questions, (3) questionnaire design, (4) sampling and evaluation of the questionnaire by means of pilot executions, (5) survey execution and, (6) analysis of results and packaging.

We designed the study with the goals of understanding:

G1 the actual diffusion of software modelling and model driven techniques in the Italian Industry,

G2 the way software modelling and model driven techniques are applied (i.e., which processes, language and tools are used), and

G3 the motivations either leading the adoption (expected benefits) or preventing it (experienced or perceived problems).

3.1.1 Research questions

Having the previously defined goals in mind, we have addressed the following research questions:

RQ1: *What is the practical relevance of software modelling and MD* in the Italian industry?*

RQ2: *Which modelling languages and notations are used in the modelling phase and for MD*?*

RQ3: *What kind of processes and tools does Italian industry adopt to support modelling and MD*?*

RQ4: *What is the level of maturity of MD* in the Italian Industry?*

RQ5: *Which are the benefits expected from modelling and MD* adoption?*

  RQ5.1: *Which are the most expected benefits?*

  RQ5.2: *Which are the relations between expectations?*

RQ6: *Which are the most frequently fulfilled expectations?*

RQ7: *Does experience in modelling improves accuracy of benefits achievement forecasts?*

The research questions from RQ1 to RQ4 concern goals G1 and G2, while research questions from RQ5 to RQ7 concern the goal G3.

The research question RQ1 concern how and why professionals use software models and apply MD* techniques in industrial software development. The research question RQ2 is about the adoption by the developers of general purpose modelling languages (e.g., UML) or domain specific languages (DSLs) and whether they prefer graphical or textual languages. The research question RQ3 aims at investigating the current tool support and adopted processes during the modelling phase.

The research question RQ4 concern G1 considering only the maturity in the usage of models and MD* for software development and maintenance.

The research question RQ5.1 is to understand which are the anticipated benefits that also represent plausible motivations for adopting modelling and MD*. By means of the RQ5.1 we envision groups of related benefits, i.e., benefits that are supposed to be achieved together. The RQ6 aims to understand how well confirmed benefits match expectations, in order to understand the capability of participants to predict the results and spot possibly hard-to-gain benefits. The RQ7 aims to understand whether (or not) experience improves (or affects) the performance of correctly forecasting achievable benefits.

3.1.2 Questionnaire Design

To design the questionnaire we adopted the Goal-Question-Metric approach [49] and followed a standard schema[60]. From the goals we derived the questions and the metrics necessary to answer them. In other words, the questionnaire has been developed to directly address the goals of the study. To maximize the number of responses we designed the questionnaire to keep the completion time within approximately 10-15 minutes [2].

The questionnaire contains 31 items formulated both as open and multiple-choice questions, though the effective number of questions actually presented to a respondent depends on his (her) level of usage of MD* and modelling (e.g., respondents non using modelling in their software process are required to answer eight questions only).

The questionnaire has the structure shown in Fig. 3.1 and consists of four sections:

Section 1 it is a section common to all respondents, and characterizes their organization. In particular, it collects: business domain, organization size, respondent's group/business unit size and experience of unit members. All the questions of this section are identified by the **Sub** prefix [3].

Section 2 this section asks about the processes adopted: the kind of projects conducted and their average duration, if the respondent uses models and with which goal. In this section the most important question is Dev08 that reads as follows: "*Are models used for software development in your organization?*". All the questions of this section are identified by the **Dev** prefix.

Section 3, Section 4 these sections are administered only to subjects who answered "yes" to question Dev08 and collect information concerning modelling and MD* usage. There are 21 questions about modelling languages, notations, tools and processes. For instance, we asked about years use of MD*, experience in MD* and percentage of projects in which models are used with respect to all the projects undertaken. Moreover, we also collected information about code generation (e.g., degree/percentage of code generated with respect to the final product), execution of models by means of specialized interpreters and usage of automatic transformations (model-to-model and model-to-text). The questions of these Sections are identified respectively by the prefix **Mod**, for the Section 3, and the prefix **Lan** for the Section 4.

3.1.3 Population and Identification of the Sample

The target population of our study consists of software development teams or business units. So as to get information about the target population we defined a framing populations consisting of Italian software professionals (e.g., project managers, architects, developers etc.) whom we asked to answer our questionnaire. To sample the population we applied both probabilistic (random sampling) and non-probabilistic (convenience sampling) methods [80].

[2]It turned out the actual time for completing the questionnaire was on average less than six minutes.

[3]A unique identifier composed by question type and number was assigned to each question

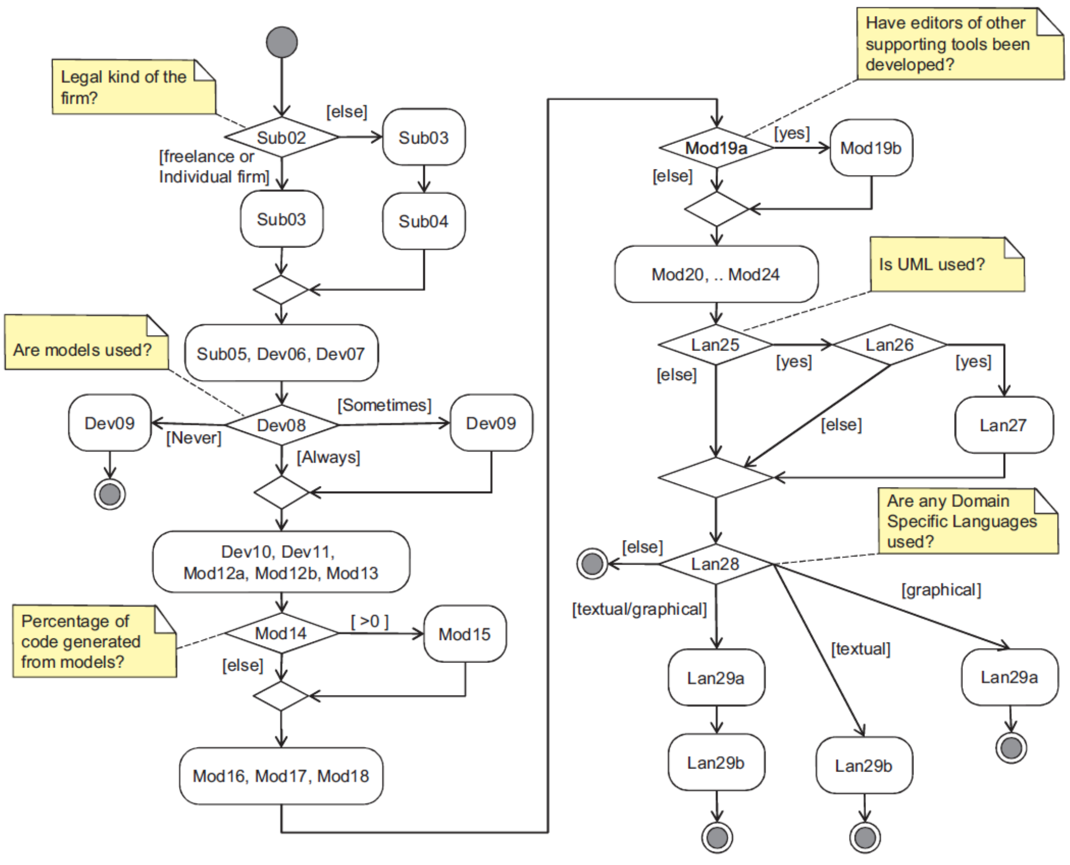


Figure 3.1: Questionnaire structure.

More in detail, the sampling was performed in five ways: (i) using the Commerce Chamber database and randomly selecting some contacts; (ii) as a convenience sampling relying on the network contacts; (iii) sending invitation messages on mailing lists concerning programming and software engineering; (iv) publishing a note on an on-line magazine for developers (programmazione.it); and (v) advertising it on a large Italian conference for developers (CodeMotion 2011).

In total, we obtained 181 responses: 155 complete questionnaires and 26 incomplete ones; the latter were discarded before the analysis phase. Thus the context of our survey consists of a sample of 155 Italian software professionals. Unfortunately, we did not know exactly how many people have been reached by our invitation messages and advertisements, so we can not calculate response rate (this kind of problem is common for on-line questionnaires, see, e.g., [84]).

It is important to emphasize that in our sample the correspondence between respondents and companies is not one-to-one. In particular, we could have more questionnaires compiled by professionals belonging to the same company but anyway employed in different business units/groups. Clearly, this is more probable in large companies hosting several business units and work groups.

3.1.4 Survey Preparation and Execution

We collected data through an on-line questionnaire created by means of the LimeSurvey [4] survey tools. The survey was put on-line from the 1st of February 2011 until the 15th of April 2011 (two and a half months). The procedure followed to prepare and administer the questionnaire, and collect the data is made up of the following five main steps:

Preparation and design of the questionnaire. First we prepared the questionnaire. Then, we conducted three different pilots with software professionals before putting on-line the final questionnaire (survey instrument evaluation phase). Pilot studies allowed us to identify possible problems with the questionnaire itself and improve the validity of the instrument [80]. In particular based on the feedback obtained, minor changes at the questionnaire were made.

On-line deployment. Once ready, the questionnaire was uploaded to the LimeSurvey survey tool to permit the automatic collection of data.

Invitation to participate. Organizations were sampled as detailed before. Once the contact persons were identified, we invited people, via email, to register to the survey and to complete the on-line questionnaire. We also broadcast invitation on selected mailing lists and on-line magazines/conferences including in the message a link (”click here to take the survey”) to a registration form where the participants could register themselves and compile the questionnaire.

Monitoring. During the data capture phase, we monitored the progress of the questionnaire submission. This allowed us to send selective reminders to contacts who did not respond or did not completed the questionnaire. Some people that reported some difficulties about the questions, because of internal policies of the company or because involved in very different projects with different companies at the same time, asked to us some clarifications about the questions.

Data analysis. After questionnaires have been collected, analyses were performed. We applied more descriptive statistics and showed our findings mainly by means of charts. Whenever possible we addressed the research questions with the support of statistical tests.

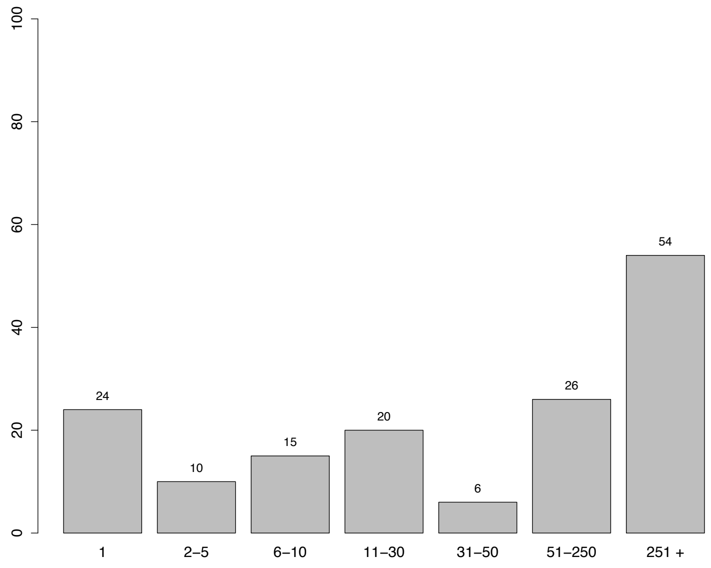


Figure 3.2: Size of respondents' companies

3.1.5 Findings

In the following we report part of the findings obtained, for the full representation of the results and a deeper analysis as well as a more complete description of the survey, refer to the published papers ([109, 108, 110]).

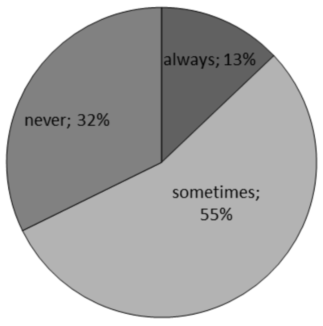


Figure 3.3: Who use modelling

The most of the companies where the respondents work are in the IT domain (104), then come

[4]http://www.iimesurvey.org/

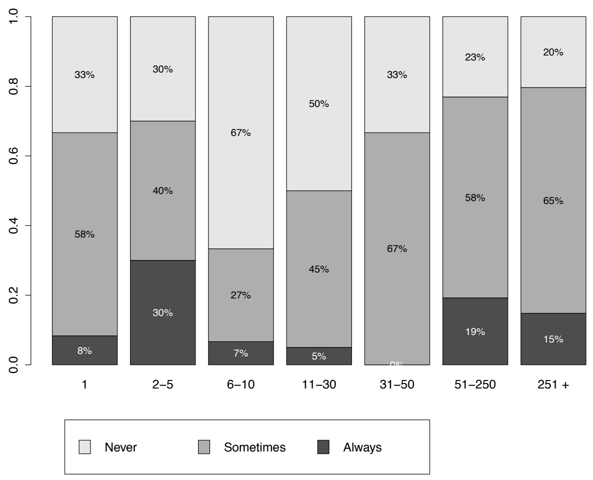


Figure 3.4: Proportion of modeling usage per company size

services (15) and telecommunications (11). The distribution of the companies size where the respondents work is presented in Fig. 3.2.

Relevance

As is shown in Fig. 3.3, among the 155 complete questionnaires, we have 20 companies (13%) always using modelling, 85 (55%) using it sometimes, and 50 (32%) never using modelling.

Such proportion varies significantly as the size of company varies, as illustrated in Fig. 3.4. We observe that the use of modelling (i.e., always + sometimes) is positively correlated with the size (i.e., it is more frequent in large companies) with two exceptions: individual professionals and micro-sized companies. Instead, the systematic use of modelling (i.e., always) is more frequent in micro and large companies; medium-sized companies and individual professionals adopt it occasionally.

Language and Notation

We found that 76% of developers (80 out of 105) creating models use UML; among them 11% use also UML profiles, 51% do not use them, and the remaining 38% state to not know if them are used in their organization. Only 21% of projects using models appear interested in Domain Specific Languages (DSLs). Among them 48% use a purely textual notation, 24% a purely graphical one, and 29% a mix of textual and graphical notations.

Processes and Tools

As far as processes are concerned we investigated which role typically performs the modelling. Usually modelling is performed by multiple roles at the same time. For this reason, to the corresponding question in the questionnaire several answers were permitted (e.g., developer and project manager). Results show that architects or project managers perform modelling in 76% of the cases, developers write models in 72% of the cases, while domain experts are involved in just 11% of the cases. As far as model manipulations, it appears the only 10% of modellers perform automatic transformations between models. While 16 of the modellers developed editors or other support tools for models. Since models can evolve, 53% of the modellers adopt versioning of models.

Level of Maturity

Taking inspiration from [90], we considered three dimensions to measure maturity:

(1) *Level of automation*; we assumed that higher levels of automation in the development process (e.g., measured as percentage of generated code) correspond to higher MD* maturity of the company.

(2) *Tools*; we assumed that the development of modelling MD* tools (such as for example graphical editors to draw models or tools for developing textual DSLs) correspond to higher MD* maturity of the company.

(3) *Acquired experience in modelling and MD**; we assumed that significant employees' experience in modelling and MD* (e.g., measured in years) equate to high maturity of the company.

Thus, we grouped the selected questions accordingly to the considered dimensions (see Table 3.1).

In the following analysis of maturity, we will consider all respondents that perform modelling, considering the sheer production of models the level zero of maturity.

Level of automation: so as to evaluate the level of automation, we considered three indicators: (1) *code generation*, (2) *model execution* and (3) usage of *model transformations*. They correspond to four items of the questionnaire (see Table 3.1). We found that 50 participants (48% of the adopters of modelling and 32% of the entire valid sample) actually employ one or more of the three key techniques.

We found that *code generation* is often used (Mod14*), indeed 46 developers, 44% of the adopters of modelling (nearly 30% of the entire valid sample), generate portions of system code in an automatic way from models. The amount of application code that is generated is 42% as mean and the median is 30%. Automatic generation targets different parts/tiers of the system (see Fig. 3.4 for the complete statistics)

Dimension	ID	Question	Type of Data	Findings
Automation	MOD14	Which percentage of code is generated starting from models?	Interval	42% of generated code
	$MOD14^*$	Derived from previous: $MOD14^* \leftarrow MOD14 > 0$	Yes/No	44% adopters of modelling
	MOD15	Which parts/tiers of the system are automatically generated? (e.g., presentation, business and data logic)	Nominal	Mostly SQL scripts, presentation logic and architectural code
	MOD16	Are models executed (interpreted) at run-time?	Yes/No	16% adopters of modelling
	MOD18	Are transformation languages (e.g., ATL) used? (model-to-model and model-to-text transformations)	Yes/No	10% adopters of modelling
Tools	MOD19a	Are specialized editors or other supporting tools developed for managing/creating models? (e.g., using Xtext that is a tool for developing textual DSLs)	Yes/No	16% adopters
	MOD19b	Which technologies are used to develop specialized editors or other supporting tools?	Nominal	GMF (6 users) and Xtext (4 users)
	MOD20	Are versioning systems used to manage models?	Yes/No	53% adopters
	DEV09	What are the problems hindering or preventing modelling and MD* (if any)?	Nominal	35% tools unavailability/immaturity/cost
Experience	MOD21	Since how many years is modelling used?	Ordinal	Median 5 years, average 5.35 years
	MOD22	In how many projects has modelling been used in the last 2 years?	Interval	Median 30%, average 44%
	MOD24	Since how many years is MD* used?	Ordinal	Median 4 years, average 4.30 years
	MOD23	In how many projects has MD* been used in the last 2 years?	Interval	Median 30%, average 39%

Table 3.1: Summary data about maturity

We counted 17 companies out of 105 that adopt *model execution* i.e., 16% among the adopters of modelling (Mod16). Adopters of *model execution* tend to generate approximately 50% of the code (median), while non-adopters have on average a code generation close to zero.

Only 11 companies out of 105 perform *model transformations*, i.e., 10% among the adopters of modelling (Mod18).

Tools: Four items in the questionnaire concern tools (see Table 3.1). They address three aspects: *development of supporting tools* (e.g., specialized editors), *versioning*, and *adoption problems*.

A total of 17 companies out of 105 have developed custom editors or other tools for managing and/or creating models. They are 16% of the adopters of modelling (Mod19a). While developing specialized editors could be an indicator that the company is willing to invest in MD* and considers it relevant, the lack thereof does not necessary imply that the MD* approach adopted is immature: the organization could adopt a ”ready-made MD* solution” developed externally.

There are 56 companies out of 105 that use *versioning* systems to manage their models, that is just 53% among all the modellers (Mod20). This percentage can be considered an element of immaturity for the community of modellers. Organizations which do not generate code, use versioning in the 46% of the cases while companies generating more than 80% of code use it in 75% of the cases.

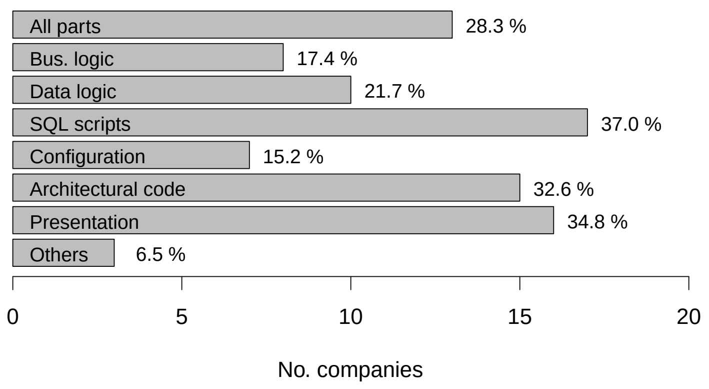


Figure 3.5: Parts/tiers of the application generated from models

Among respondents who reported problems hindering or preventing modelling adoption (Dev09), 35% of them reported one or more problems related to supporting tools . Reasons of dissatisfaction about the tools are: their unavailability (23 respondents, 17% of the ones reporting problems) or immaturity (14, 10%), their cost (14, 10%) or fear of vendor lock-in (13, 9%). Out of 31 respondents which complained about the available tools, four of them developed some tools in-house.

Experience: Four items in the questionnaire are devoted to the experience dimension (see 3.1). They capture information regarding modelling and MD* employees experience measured in years (*staff experience*) and percentage of projects where modelling and MD* are adopted respectively (*organization experience*). The responses are summarized in Fig. 3.6.

Regarding the *staff experience*, the Fig. 3.6 (middle) shows the complete distributions for experience in modelling and MD*. The experience in modelling appears to be distributed according to a normal distribution centred around the interval of (2,5] years of experience. Instead, the experience in MD* has a distribution strongly skewed towards the zero.

For the *organization experience* we can see that both modelling and MD* are used in more projects as the experience of the respondents in the field grows (see Fig. 3.6, up for modelling and down for MD*). For example, see Fig. 3.6 (up), modellers with an experience in the range (0,2] years adopt modelling only in the 20% of the projects (median) while modellers with more than 10 years of experience adopt it in the 80% of the projects (median). The correlations between years of experience and proportion of projects adopting modelling and MD* respectively are statistically significant. In both cases the Kruskal-Wallis test returned a p-value <0.001.

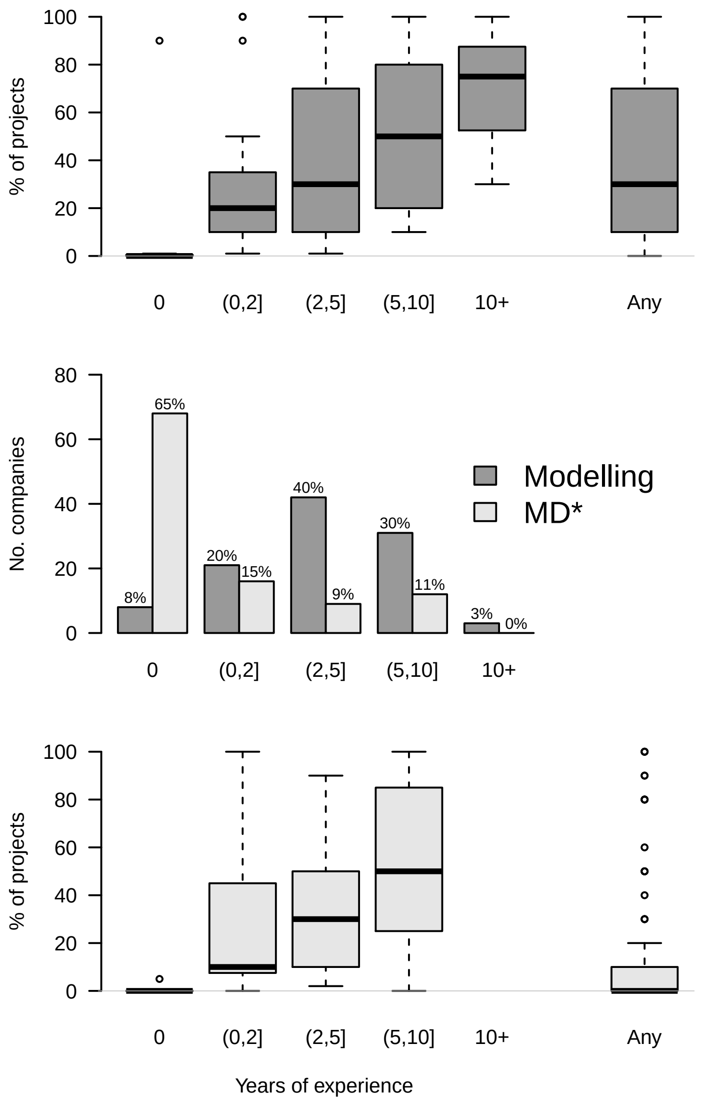


Figure 3.6: Staff (middle) and Organization experience (up and down) in modelling and MD*

Expected Benefits

An initial item (Dev08) concerned whether models are used in the respondent organization for software development. For the respondents who provided a positive answer to such item we administered a further item measured using the question "*What are the benefits expected and verified from using modelling (and MD*)?*". This was designed as a closed option question; the list of benefit that we presented the respondents was compiled on the basis of the literature and includes:

- Design support
- Improved documentation
- Improved development flexibility
- Improved productivity
- Quality of the software
- Maintenance support
- Platform independence
- Standardization
- Shorter reaction time to changes

For each benefit the respondent could indicate whether the benefit was expected and/or verified. To evaluate the experience in modelling we considered one item that measured the years since the initial adoption of modeling or MD* in the work-group of the respondents.

Here we consider first results about the RQ5.1 and then results about the RQ5.2. Regarding the RQ5.1 we can see the Table 3.2, where we report for each benefit the frequency of expectation (column Freq.) and the corresponding percentage of respondents (column Estimate). *Improved documentation* is the most expected benefit, with almost 4 out of 5 respondents anticipating it. Also *Design support*, *Quality of the software*, *Maintenance support*, and *Standardization* are frequently expected. For all of the top 5 benefits we are 95% sure that more than 50% of modelling adopters expect them: in fact the confidence interval (C.I.) lower bounds are larger than 50%. The remaining benefits, *Improved development flexibility*, *Improved productivity*, *Shortened reaction time to changes*, and *Platform independence* are less popular, with the latter typically expected by less than 40% of respondents.

Regarding the RQ5.2 we report the statistically significant relations among benefits in the graph shown in Fig. 3.7: the nodes represent the individual benefits, the edges represent a statistically

Benefit	**Freq.**	**Proportion Estimate**	**Proportion 95% C.I.**	**Fulfillment Rate**
Improved documentation	81	77%	(68%, 85%)	68%
Design support	77	73%	(64%, 81%)	78%
Quality of the software	75	71%	(62%, 80%)	49%
Maintenance support	66	63%	(53%, 72%)	52%
Standardization	64	61%	(51%, 70%)	52%
Improved development flexibility	51	49%	(39%, 58%)	45%
Improved productivity	42	40%	(31%, 50%)	45%
Shorter reaction time to changes	41	39%	(30%, 49%)	37%
Platform independence	32	30%	(22%, 40%)	34%

Table 3.2: Frequency of expectations

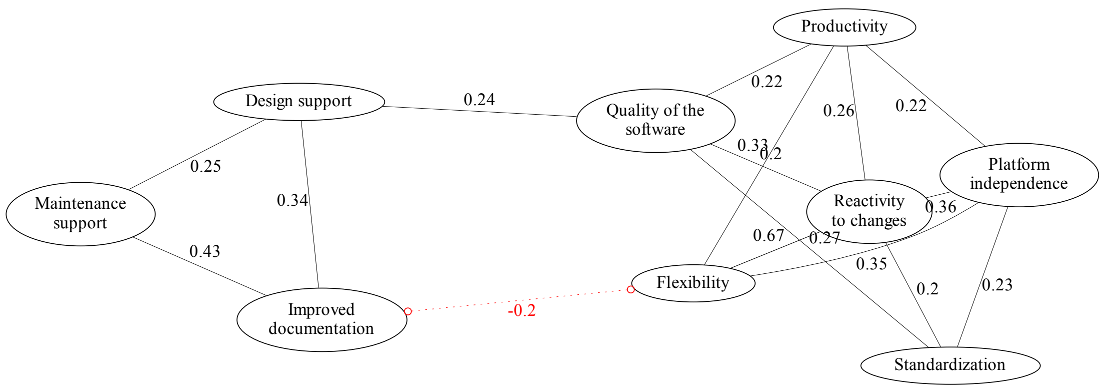


Figure 3.7: Relations among benefits expectations.

significant relation which is reported as edge label. The layout of the nodes is computed considering the Kendall rank correlation coefficient (KC) (the length of the edge should be as much as possible inversely proportional to the Kendall distance) and additional constraints to improve the readability avoiding the overlaps of nodes and labels. The benefit expected together (KC >0) are linked by continuous black lines, while the benefits whose expectations tend to exclude each other (KC <0) are linked by dashed lines, with circles at the ends. All the significant relations were positive except one, that between *Improved documentation* and *Improved development flexibility*: who expects one of these two benefits tend to not expect the other one.

By observing Fig. 3.7, we can note two distinct clusters: the first includes *Improved documentation*, *Design support* and *Maintenance support*. The second one includes *Improved development flexibility*, *Shorter reaction time to changes*, *Platform independence*, *Standardization* and *Im-*

proved productivity. *Quality of the software* appears to be a transversal benefit, connecting the two clusters.

Most Frequently Fulfilled Expectations

The research question RQ6 concerns how often the verification of a benefit met the expectation. It is measured as the frequency of verified benefit given the benefit was expected. Results are reported in the rightmost column of Table 3.2 (Fulfilment rate). *Design support* has the highest fulfilment rate: 60 respondents out of the 81 who reported to expect it (i.e., 78%) actually achieved the benefit. Also *Documentation improvement* is consistently verified when expected, the same is not true for all the other benefits. *Standardization* and *Maintenance support* are just above the parity (it means are slightly mainly achieved than not achieved, when expected) and all the others are more often not achieved than achieved. *Platform independence* and *Reactivity to changes* have a really low fulfilment rate, representing very often a delusion for practitioners.

Improvement in the Accuracy of Benefits Achievement Forecasts

Concern the RQ7:"*Does experience in modelling improves accuracy of benefits achievement forecasts?*", we can say that the low experienced practitioners group (<5 years of experience in modelling) is constituted by 50 persons, whereas the high experienced practitioners group (i.e., $\geq$ 5 years of experience in modelling) by 55. Thus, the two groups are balanced.

Applying the Fisher test to the built contingency table, even adopting a looser threshold of 0.1, it is not possible to find any statistically significant difference. Therefore, we conclude that experience does not improve the precision in forecasting the obtainable benefits.

3.1.6 Debriefing Session

After the data analysis and interpretation of the results, we conducted a debriefing session with three expert software professionals, which participated in our survey, so as to understand MD* findings that are difficult to interpret. The experts we asked for clarification cover different features of MD*: (1) is the responsible architect for the design of an in-house MD* solution (in short, a suite for the rapid development of information systems) for a large organization; (2) is the CEO of a company producing a MD* tool for the development of Web applications based on code generation; (3) is the Sales & Marketing Director of a company producing a model driven Web application framework based on run-time execution of models.

Moreover, we took advantage of the availability of such qualified professionals and asked them what they believe is needed to improve the maturity of MD* in the Italian industry.

Experience in MD* is very low Low experience in MD* has been attributed to several different causes: mostly the fact that apparently models are used as tools for documentation or analysis and not as artefacts inserted into a MD* approach. According to our experts, the primary cause for this is little popularity in the industry that limits the capability for developers to build on-thejob experience. One of the professionals also suggested that huge standardization effort, that bring countless notations and techniques together, may result intimidating and actually preventing diffusion of MD* practices. The considerations from the experts strengthened our consideration about the opportunities in the industry and academia.

The percentage of code that is generated is often low Here, returns the supposed use of models barely for documentation purposes or for generating just the initial code skeletons; the typical case consists of generating the code structure from a UML class diagram. In our experts opinion, the primary cause for this limited use of models lies in the scarcity of appropriate tools and the limited knowledge of the few available. In practice, limited tools are used at individual developers level because of several factors: it is difficult to get management commitment at team or organization level, common modelling languages (e.g., UML) allow just a limited code generation, and aim for more extensive generation clashes with the fear of losing control over code.

What is needed to improve the maturity and foster the diffusion of MD* in Italy? The experts mentioned factors in three categories: languages, tools, and processes. Standardized languages are the key to the diffusion of MD* approaches; UML and BPMN are positive examples but are not sufficient because they do not cover all the relevant aspects (e.g. interactions and systems communication). Moreover, MD* usage requires integrated tool-sets supporting the full development process. From a process perspective, there is a need for customized processes that include not only the generation but also, release management, versioning, and deployment. As far as management is concerned, a successful application of MD* techniques requires understanding the key success factors, the applicability in different domains and the skills required from developers. From a more general perspective, focus on quantitative aspects of software production does not incentive use of models, which can be exploited when quality is considered. In addition, at a management level, it is important to know which are the success factors for different domains and the skills required to practice MD* techniques.

3.1.7 Threats to validity

We analyse the potential threats to the validity of our study according to the four categories suggested in [116].

Construct validity the questions we asked are very simple and straightforward therefore we are confident that they actually measured what they were intended for.

Internal validity the main issues affecting the internal validity of our study concern the framing and sampling of the participants.

- We incurred in a possible selection bias due to the self exclusion of participants not interested in modelling. Self exclusion is a well-known problem especially in Internet surveys advertised by means of mailing lists and groups. The possible threat consists in an over estimation of the proportion of respondents who declared interest in modelling and therefore of the overall relevance of modelling and MD* in the Italian industry. However, we believe that the aspects of maturity that we analysed (i.e., level of automation, tools, and experience) are not affected by this threat.
- Another threat derives from the possible foreign units in the sample: the target population of our study consisted of development teams, it is possible that the questions were answered by a responded without the required knowledge. We addressed this concern in the protocol: we explicitly required the questionnaire to be filled in by technical personnel involved in the development. Even in case of a knowledgeable respondent, he/she could be unaware of some details; clearly this is more likely if the team is very large.
- Eventually, the sampling procedure made it possible to select duplicate units: two different members of the same development team could have answered our questionnaire. We addressed this threat by means of a post-survey validation: we found that the respondents from the same company actually worked in distinct business units and belonged to distinct teams. For this reason, we decided to consider them in the valid sample.

Conclusion validity most of our analysis was based on simple descriptive statistics, in the few cases where hypothesis testing was opted for non-parametric tests that can be used without specific assumptions.

External validity we used a non-probabilistic sampling schema since we expected a low diffusion of MD* techniques in the industry and consequently we supposed it was difficult to contact a reasonable number of adopters. This should be considered interpreting the results we obtained: even if the demographics of our sample is quite diverse, the generalization of our results to the entire population may not be appropriate. Moreover, given the sampling strategy we adopted we can not calculate the response rate.

3.2 An Experimental Assessment of Maintenance Tasks with Model Driven Techniques

In the context of our study about the effectiveness of model driven development we have done a controlled experiment with 21 bachelor students aimed at investigating the effectiveness of model driven development during software maintenance and evolution activities. Here we report some of the results obtained, other results and more information can be found in [95].

Obviously, not all the development methods and related tools can be experimented with respect to every software characteristic (e.g., portability, maintainability and interoperability). Thus, we had to select an instance among all the possible MDD proposals and to select a characteristic. Finally, after having consulted the literature looking for non proprietary MDD tools, we resort to UniMod [72] and maintainability as main software characteristic to study.

Thus, our main research question can be stated as follows:

Is UniMod effective during software maintenance tasks? And, if yes, how much is UniMod effective?

UniMod is a MDD method for designing and implementing object-oriented programs. The method relies on a specific instance of automata-based programming (SWITCH technology [101]) and on UML. A tool for the development and execution of UniMod models is available. The tool allows one to create, edit and execute UML class diagrams and state machines.

We chose it for three reasons: (1) the method relies on UML that is well-known by our students, (2) a MDD free tool [5] (available also as an Eclipse plug-in) based on this method exists, (3) the UniMod tool is simple to use and to install and thus suitable for students.

3.2.1 Experiment Definition, Design and Settings

We conceived and designed the experiment following the guidelines by Wohlin et al. [115]. The mind map shown in Fig. 3.8 summarizes the main ingredients of the controlled experiment and the obtained results.

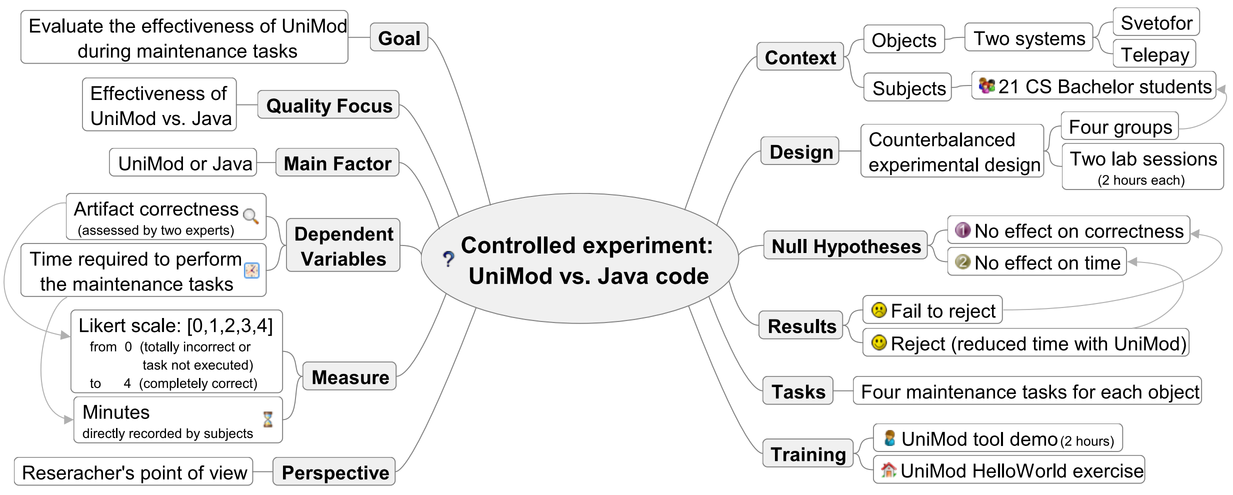
The goal of the study is analysing the effectiveness of UniMod (i.e., method + tool) during maintenance tasks with respect to code-centric conventional programming (i.e., directly implementing the change requests in the code using an IDE). The perspective is of researchers, evaluating how effective is UniMod in terms of (1) correctness of the final artifacts and (2) time required to perform the maintenance tasks.

Hypotheses

The null hypotheses for the study are the following:

- H_0a Using UniMod does not affect the correctness of the maintained artefacts.

[5] `http://unimod.sourceforge.net/`



Controlled experiment: UniMod vs. Java code
Goal
Evaluate the effectiveness of UniMod during maintenance tasks
Quality Focus
Effectiveness of UniMod vs. Java
Main Factor
UniMod or Java
Dependent Variables
Artifact correctness (assessed by two experts)
Time required to perform the maintenance tasks
Measure
Likert scale: [0,1,2,3,4] from 0 (totally incorrect or task not executed) to 4 (completely correct)
Minutes directly recorded by subjects
Perspective
Reseracher's point of view
Context
Objects
Two systems
Svetofor
Telepay
Subjects
21 CS Bachelor students
Design
Counterbalanced experimental design
Four groups
Two lab sessions (2 hours each)
Null Hypotheses
No effect on correctness
No effect on time
Results
Fail to reject
Reject (reduced time with UniMod)
Tasks
Four maintenance tasks for each object
Training
UniMod tool demo (2 hours)
UniMod HelloWorld exercise

Figure 3.8: Mind Map of the experiment

- H_0b Using UniMod does not affect the time required by the maintenance tasks.

H_0a and H_0b are two-tailed since we cannot postulate a difference expectation in terms of correctness and time.

Treatment and Experimental Design

Two cases can be distinguished: (i) maintenance tasks executed on Java code using the Eclipse IDE and (ii) maintenance tasks executed on the UniMod model using the Eclipse IDE with installed the UniMod plugin. Thus, only one independent variable occurs in our experiment (main factor), which is nominal. It assumes two possible values: (Java only) or + (UniMod).

Table 3.3: Experimental Design (**–** = Java only, **+** = UniMod).

	Group A	**Group B**	**Group C**	**Group D**
Lab 1	Svetofor **+**	Svetofor **–**	Telepay **–**	Telepay **+**
Lab 2	Telepay **–**	Telepay **+**	Svetofor **+**	Svetofor **–**

The experiment adopts a counterbalanced design (see Table 3.3), thus ensuring that each subject works with both the treatments (UniMod and Java only) in the two lab sessions (2-hours each).

Subjects were split randomly into four groups, each one working in Lab 1 on the maintenance tasks of a system with a treatment and working on Lab 2 on the other system with a different treatment. This design was chosen because it mitigates possible learning effects between labs (Lab1 and Lab2).

Objects

The objects of the study are two software systems: *Svetofor* and *Telepay*. The UniMod versions of these two systems were downloaded from a Website managed by the UniMods designers. Starting from the two downloaded UniMod projects, we completely built in Java two new software systems (Svetofor and Telepay) equivalent to Svetofor+ and Telepay+. During the transformation, we chose to implement the state machines using the nested switch approach [65].

Svetofor is a smart traffic lights simulator equipped with a GUI representing pedestrians and cars. The user can click the buttons on the right to make pedestrians and cars appear in the GUI. The system simulates a smart traffic light that works as follows. If there are no cars, but some pedestrians are waiting on the crossing, the traffic light indicates green light for pedestrians until at least one car appears, and vice versa. If there are neither cars nor pedestrians, the green light is for cars (they need more time to slow down and speed up).

TelePay simulates a mobile phone payment terminal. The user can use the keyboard to insert the phone number. Then, she/he can select some banknotes to charge the mobile phone.

Subjects

The subjects were 21 students from the Software Engineering course, in their last year of the bachelor degree in Computer Science at the University of Genova (Italy). The subjects had a good programming knowledge, in particular Java programming, and an average UML knowledge (which was explained during the course).

Variables

The dependent variables to be measured in the experiment are (i) the artefact correctness after the execution of the maintenance tasks and (ii) the time required to perform the tasks. The time was measured by means of time sheets. Students recorded start and stop time for each implemented maintenance task. In this way, we were able to compute the time required to execute each maintenance task simply computing *stop time* - *start time*. Finally, the time variable is computed for each subject summing up these four values.

Procedure and Execution

For each lab, the subjects had two hours available to complete the four maintenance tasks: CR1-CR4. Between the two laboratory sessions (Lab 1 and Lab 2) a break was given. For each Lab session, the experiment execution followed the steps reported below:

1. We delivered a sheet containing the description of the system.
2. Subjects were given 10 minutes to read the description of the system and understand it.
3. Subjects had to write their name on the delivered sheet.
4. Subjects had to download from a given URL the Eclipse project and import it.
5. A sheet containing the four change requests was delivered. For each change request (CR1-CR4):
   (a) Subjects had to satisfy the change request (for Svetofor or Telepay).
   (b) Subjects had to record the time they need to execute the modification (start/stop time).

3.2.2 Results

Results of statistical tests are considered to be significant for a significance level of 95%.

In Fig. 3.9 is summarized the distribution of the **correctness** variable by means of boxplots. “In descriptive statistics, a box plot or boxplot (also known as a box-and-whisker diagram or plot) is a convenient way of graphically depicting groups of numerical data through their five-number summaries: the smallest observation (sample minimum), lower quartile (Q1), median (Q2), upper quartile (Q3), and largest observation (sample maximum). A boxplot may also indicate which observations, if any, might be considered outliers”.[6]

Observations are grouped by treatment (UniMod or Java) and shown partitioning by system (Svetofor and Telepay). The y-axis represents the cumulative correctness of the four maintenance tasks the cumulative score 16 represents the maximum value of correctness and corresponds to four tasks completely correct. The boxplots show that the subjects achieved a better correctness level when accomplishing the tasks with UniMod. We test the overall difference (i.e., without partitioning by system) in correctness by applying an unpaired non parametric statistical test.

By applying a two-tailed Mann-Whitney U test, we found the difference not to be statistically significant (p-value=0.53), therefore we cannot reject the first null hypothesis (H_0a).

[6]wikipedia: `http://en.wikipedia.org/wiki/Box_plot`

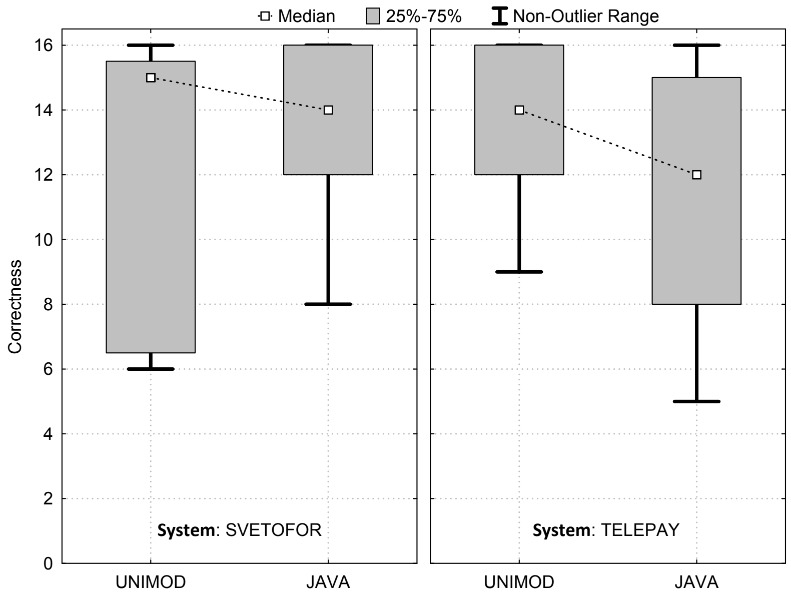


Figure 3.9: Boxplots of Correctness

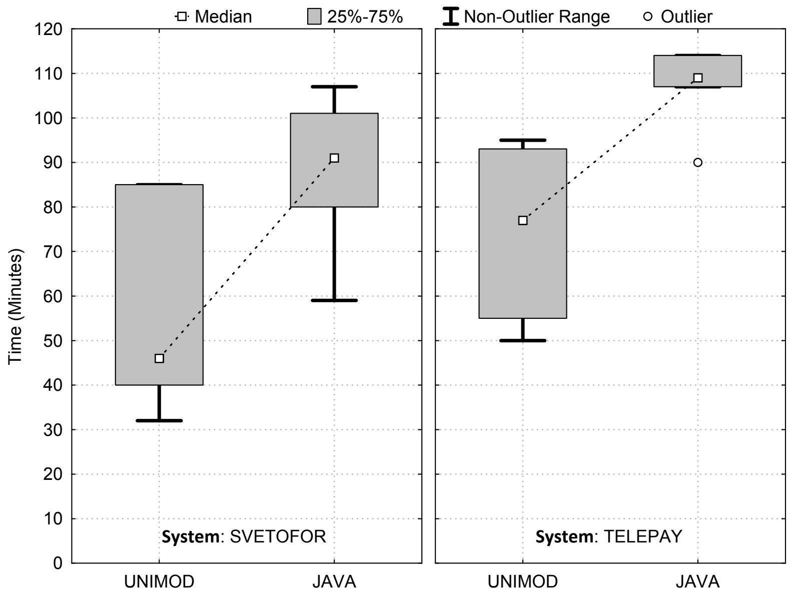


Figure 3.10: Boxplots of Time

The second hypothesis can be easily tested by looking at the **time** required to complete all the tasks. For this reason, we analysed only the time for the students able to complete all the four tasks (25 observations in total). In Fig. 3.10 is summarized the distribution of the time variable by means of boxplots. The boxplots show that, for both systems, students using code-centric programming (i.e., Eclipse + Java) employed more time than students using UniMod. The difference between the two treatments is more evident for Svetofor. Overall, the mean time reduction, achieved with UniMod, is 33 minutes (with UniMod the mean is 65 minutes while with conventional programming it is 98 minutes), i.e., approximately 33,7%.

To test the second hypothesis, we used a Mann-Whitney U test as done for the first hypothesis. The results of the two tailed Mann-Whitney test show that the overall difference is statistically significant (p-value <0.01). Therefore, we can reject the second null hypothesis H_0b.

3.2.3 Threats to Validity

Internal validity threats concern external factors that may affect a dependent variable (in our case code correctness and time). Since the students had to perform a sequence of four maintenance tasks, a learning effect may intervene. However, the students were previously trained, the tasks were of progressively increasing difficulty, and the chosen experimental design (with a break between the two labs) should limit this effect. Therefore, we expect learning not to have influenced too much the results.

Another threat could be due to a possible level of subjectiveness caused by the construction of the Java versions of Svetofor and Telepay that we did starting from the UniMod projects. We applied a standard approach to implement state machines in Java (in practice, we used nested switch [65]) but we cannot be sure that changing the implementation (e.g., using the state pattern) the outcome of the experiment would not be affected. We plan to verify this as part of our future work.

Construct validity threats concern the relationship between theory and observation. It is possible that the used test suite does not provide an adequate means to measure the quality of the change requests implementation. Moreover, given that the test suite was manually executed, some errors could have happened in the evaluation process. We mitigated this last possible threat with an independent double check.

Threats to *conclusion validity* can be due to the sample size (only 21 students) that may limit the capability of statistical tests to reveal any effect. We chose to use non parametric tests due to the size of the sample and because we would not safely assume normal distributions.

Threats to *external validity* can be related to: (i) the choice of simple systems as objects and (ii) the use of students as experimental subjects. We do not expect the absolute performance of students being at the same level of professionals, but we expect to be able to observe a similar improvement trend. Further controlled experiments with larger systems and more experienced developers are needed to confirm or contrast the obtained results. Another threat to external validity is that the results are limited to UniMod, and rather different results could be obtained with other MDD approaches.

3.3 Conclusion

In this chapter we described our survey designed and conducted to study the diffusion of modelling and model driven techniques. Here, we reported some of the results described in our works [109, 110, 108].

Moreover, we reported the results obtained done a controlled experiment to investigate the impact of following a model driven approach during software maintenance and evolution activities. Also these results are described in our work [95].

Chapter 4

The Precise Modelling Approach

In this chapter we present the *Precise Approach* to modelling using the UML specific items for specific aims. As we will see in the next chapter, models written following this approach are those that will be used as source of model transformations developed using the *MeDMoT*

First, in Sect. 4.1 the Precise Modelling Approach is introduced, then we present three applications:

– in Sect. 4.2 we describe MRelational a way to model relational databases using the UML,

– in Sect. 4.3 we describe MOntology a way to model ontologies using the UML, and

– finally in Sect. 4.4 we describe MJavaD a way to model the design of Java desktop applications, again using the UML.

4.1 Precise Modelling

Modelling with the UML may be made in a quite free way producing also incorrect UML models (i.e., not conforming to the official specification), with natural language inscriptions (with all the problems of using natural language text for specifying/modelling) [94, 93]. Models suitable for being transformed must not have ambiguities, inconsistency or like, thus they cannot freely made. Precise modelling can guide modellers to write models without these problems, and then suitable for being transformed. This approach aims to build modelling methods which give guidelines to write “correct” models, hiding formalities but keeping disciplined rigour of formal methods [46].

The precise approach prescribes that in any modelling task we should determine a class C of UML models, or more precisely a class of model written using a specific UML profile, whose form is precisely defined, by means of a (conceptual) metamodel and by a set of well-formdeness rules. This will help the modeller to avoid common mistakes, to produce models of high quality, and to avoid to loose time to decide which of the huge number of the available UML constructs should be used in the specific case.

The syntax (form) of the UML models is defined by means of a metamodel and by a set of well-formedness constraints, see [68]; however this definition is not too stringent and so a lot of UML models which are acceptable for the official definition are however questionable. Moreover, UML does not have a rigorous (neither static nor dynamic) semantics, that in some part appears obscure, or at least subjected to more than one interpretation. Thus, the precise modelling approach proposes to use in any case only a subset of the UML defined by imposing further well-formedness constraints. The elements composing this subset are a selection of those UML constructs whose semantic is sound. In Sect. 4.1.1 we present this subset.

The UML allows to build models of different level of rigour, from quite informal models, where the textual inscriptions on the diagrams are natural languages fragments, to precise models where instead the OCL and the *UML action language* are used. OCL is a formal language used to specify constraints over UML model elements such as classes or operations (see [69]). Moreover, it is a pure functional language, thus OCL expressions are completely without side effects. The UML “actions” are the basic one required by an object-oriented programming language: operation call, creation and deletion of objects, assignment, and control flow statements (conditional and while-loop).

A specific case of a precise modelling method will then also clarify the level of rigour of the models, depending on what is the assumed usage of the models to be produced, think, e.g., if they will be used as a starting point for automatic code generation or as a pure communication means among non UML-literate people. Moreover, it must clarify which is the intended use of the produced model and what entities are modelled.

Summarizing, to propose a modelling method following the precise approach we need to give:

1. an informal description of the items to be modelled (a natural language text);
2. a description of the intended use of the produced models (again a natural language text);
3. a UML profile[1];
4. a (conceptual) metamodel determining the form of models to be produced, represented by means of a UML class diagram including a class named as the modelled items;
5. a set of well-formedness rules over the models characterized in 4., technically they are constraints on the class diagram defined in 4.;

[1]A UML profile is a set of extension elements (stereotypes, tagged values and constraints) that support the profiling mechanism [66]. It allows, by means of specialization, to to add semantics and constraints to the UML metamodel so to adapt the UML language to the chosen domain.

6. guidelines to drive the modeller to produce the models defined by 3., 4., and 5.

The stringent requirements on the form of the models should help the modellers to avoid the most common mistakes and to save time, avoiding to think which UML constructs to use and which is the best way to use them. Some well-formedness rules concern with the quality of the models. The purpose of these rules is to check if a model is conform to some quality criteria (i.e., comparing the measurement of some metrics with a predefined values). In some sense the formalized precise form of the models is a way to allow some best practices to be transferred by experts to the modellers.

4.1.1 The Precise UML

Here we present the subset of UML to be used by the precise modelling methods, and its intended semantic meaning, for simplicity we considers only the diagrams used in the case studies presented in this thesis, i.e., class diagrams, object diagrams, and state machines (in [93] you can find the definition of the precise activity diagrams.

A precise UML model is made of class diagrams, object diagrams, and state machines built using the following constructs:

Class diagram: classes with attributes, operations and methods, datatypes and interfaces; specialization, aggregation and composition relationships, and user defined binary associations. Moreover, invariant constraints may be associated with classes and datatypes, and pre-postconditions with operations.

Object diagram: objects with slots, and links among objects.

State machine: built using states, transitions among states, and , starting and final pseudo-states. The events may be only call events or destroy or create, and the transition effects should be UML action expression. The operations of the context class used as call event in the state machine have no associated methods; for what concerns the semantics we assume that the event queue[2] is handled following a FIFO policy

Moreover, all expressions, conditions and constraints appearing in any diagram are expressed by using OCL.

All the produced UML models must be statically correct, or, using a UML-community terminology, statically consistent, and thus satisfying all the "Constraints" listed in the official reference specification [25], plus some following additional one, defined by ourselves. We summarize all the relevant constraints for the considered diagrams below.

[2]The event queue is the location where the events are placed when they are received and wait to be processed by the state machine

Class Diagram

- A type used for typing attributes, operation parameters and results must be either a UML Primitive type[3] or an OCL basic type (see [69]) or a datatype defined in the model[4].
- The transitive closure of the specialization relationship has no loop.
- A (pre/post/invariant) constraint consists of an OCL expression that is correct w.r.t. the considered class/datatype.
- A class/datatype can have either zero or one invariant;
- An operation can have zero or one pre-condition.
- An operation can have zero or one post-condition.
- Each formula used in a constraint must be satisfied in the model.
- Two or more classes belonging to the same inheritance tree must have the same stereotypes.
- A class has an associated state machine (describing its behaviour) if and only if the class is active[5].
- An operation belonging to a datatype must have a return value and must be a query.

Object Diagram

- A link must be an instance of some association in the class diagram.
- An object must be an instance of some class C in the class diagram, and must have a slot for each attribute of C.

State Machine Diagram

- All call events are built using operations of the context class.
- The guard of a transition must be an OCL expressions that has type Boolean and is well-formed w.r.t. *self* typed by the context class of the state machine and the parameters (if any) of the event of the transition.

[3]UML primitive types are defined in [68]. They are: Boolean, Integer, UnlimitedNatural, and String.

[4]A stylistic constraint to force the use of UML associations in the class diagram, thus structural relationships between classifier are clearer.

[5]An active class owns an execution thread, hence active objects, instance of active classes, can initiate control activity [63].

- Each transition in a state machine, except the one leaving the initial state, must have an event.
- The effect of a transition must be a UML action well-formed w.r.t. *self* typed by the context class of the state machine, and the parameters (if any) of the event of the transition.

4.2 MRelational: a Simple Precise Modelling Method

MRelational is a toy precise method concerning the modelling with the UML of the relational databases.

In this case, thus the *modelled entities* are the relational databases[6], or better their schemas.

These models *will be used* for designing the databases, and later for generating automatically the SQL code for implementing them.

To model a database schema MRelational chooses to *represent*:

- database *tables* with classes stereotyped by ≪table≫;
- database table attributes with UML class attributes. For simplicity MRelational allows to use only attributes typed by UML primitive types and forces the attribute multiplicity to be equal to one;
- relations between tables with UML binary associations between classes,
  - *one-to-many* relations with directed associations with unbound multiplicity on the navigable end,
  - *many-to-many* relations with binary associations with both sides navigable, and with unbound multiplicity at both ends;
- table *primary keys* with attributes stereotyped by ≪key≫.

Classes representing tables may be related by specialization relations. Foreign key constraints are not explicitly modelled (they are a way to implement one to many relations).

Fig. 4.1 shows the conceptual metamodel for the models that represent a database schema.

[6]see `en.wikipedia.org/wiki/Relational_database`

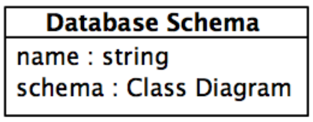


Figure 4.1: MRelational Conceptual Metamodel

The UML Profile

The UML profile used to model database schemas has then just two stereotypes: ≪table≫, a stereotype of class (classes that have this stereotype represent tables); and ≪key≫, a stereotype of attributes (attributes that have this stereotype belong to the primary key of a table).

The Well-Formedness Rules

Let *DBM* be a model of a database schema, and let *SCH* = *DBM*.schema be the contained class diagram (see Fig. 4.1).

- *SCH* cannot contain datatypes;
- *SCH* cannot contain interfaces;
- *SCH* cannot contain abstract classes;
- all the classes in *SCH* must
  - be stereotyped by ≪table≫,
  - have at least one attribute stereotyped by ≪key≫, considering also the attributes of the specialized classes (if any);
  - have only attributes with multiplicity equal to one, and and typed with UML primitive types (Boolean, Integer, UnlimitedNatural, String and Real);
- all the associations in *SCH* are binary;
- all the associations in *SCH* are named, while the ending roles are unnamed;
- all the directed associations in *SCH* must have the ending role with multiplicity *;
- all the bi-directional associations in *SCH* must have both association ends with multiplicity *.

Fig. 4.2 shows a possible (and valid) class diagram representing a database schema following the MRelational method.

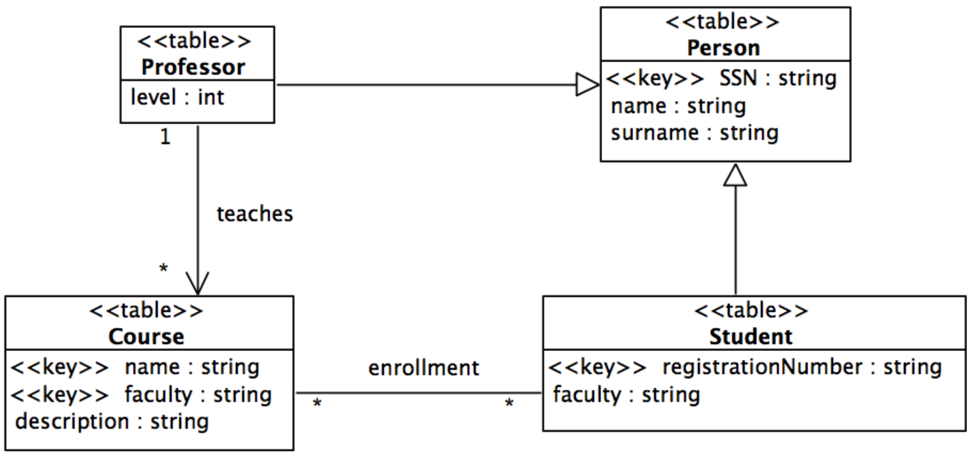


Figure 4.2: MRelational Example Model

4.3 MOntology: Using the UML to model ontologies

Here we present MOntology, a precise method to model ontologies using the UML.

The *modelled entities* are in this case the ontologies[7], and their *models are used* for designing/producing them, and later for generating a corresponding implementations using the OWL language[8] with the RDF/XML concrete syntaxes.

To model an ontology MOntology chooses to *represent*[9]:

- classes with UML classes stereotyped by ≪category≫, or classes stereotyped ≪instances≫ in this case individuals and classes are modelled at same time;
- individuals with UML objects;
- attributes with class attributes and slots;
- relationships with associations and links;
- restrictions with association and attribute multiplicities.

The form of our UML models representing the ontologies is shown in Fig. 4.3 by means of a conceptual metamodel, represented by a UML class diagram.

[7]see en.wikipedia.org/wiki/Ontology_(information_science)
[8]see www.w3.org/TR/owl2-overview/
[9]We refer to the ontology components and not to OWL constructs representing these ontology components

An ontology model is composed by a class diagram (the StaticView) defining the structure of the ontology in terms of categories, specializations, and associations, and possibly by an object diagram (the InstancesView) describing the individuals of the ontology (e.g., which are the individuals, the values of their attributes and the links among them). Obviously, the object diagram must be defined with respect to the class diagram, i.e., its instances must be typed by classes present in the latter, and similarly its links and slots must correspond respectively to associations and attributes in the latter.

The UML class diagram and the object diagram may be split in several ones, reminding that in the UML the same class/instance may appear in several diagrams, and that is not mandatory that all its features appear in each diagram.

The UML Profile

The UML profile used to model ontologies is composed by two stereotypes: ≪category≫ and ≪instances≫. A class stereotyped by ≪category≫, will represent a category of the ontology, its individuals will be defined in the instances view. ≪instances≫ stereotypes enumeration datatypes; an enumeration stereotyped by it will define simultaneously a category and its individuals, defined by its literals.

The Well-formedness rules

Let sW and isW be respectively the StaticView and the InstanceView of an ontology model, as defined in Fig. 4.3.

- every class in sW must be stereotyped by either ≪category≫ or ≪instances≫;
- every association in sW must have named ends and must be anonymous;
- two association ends in sW cannot have the same name;

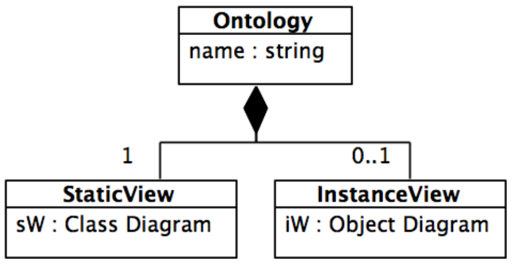


Figure 4.3: MOntology conceptual metamodel

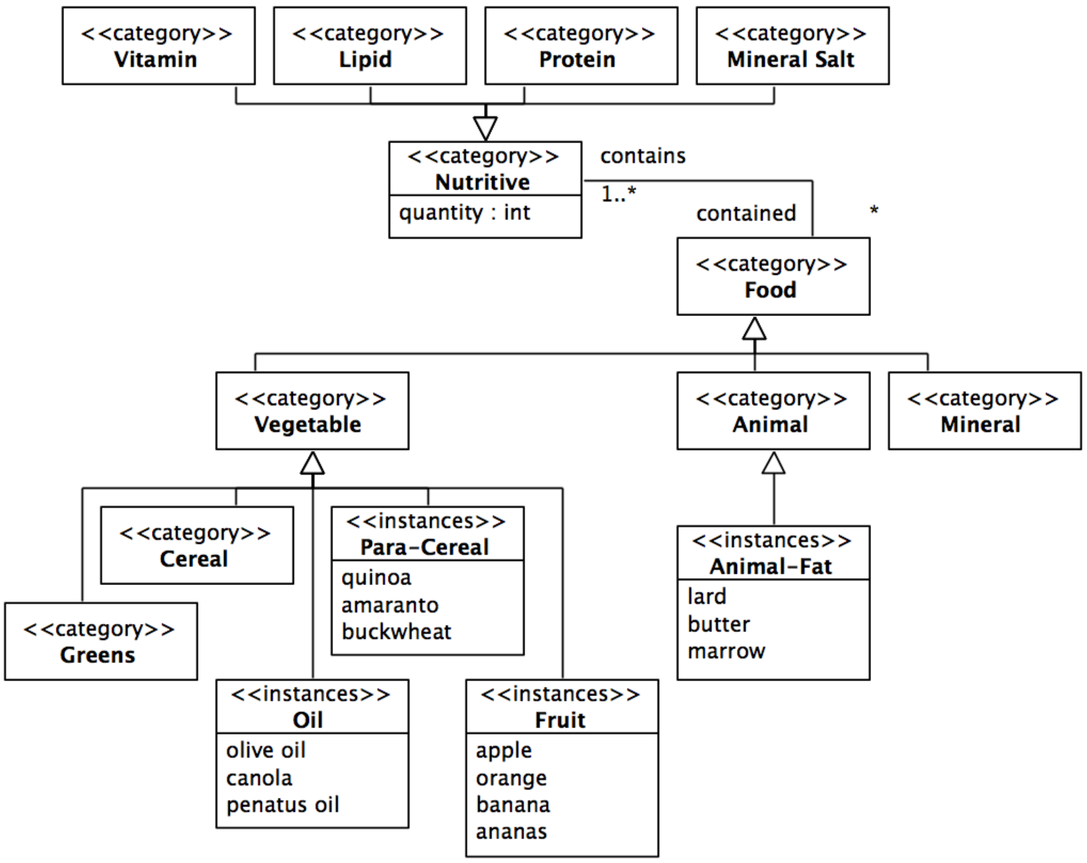


Figure 4.4: MOntology Food Ontology fragment

- two attributes also if of different classes in sW cannot have the same name;
- every attribute of a class in sW must have multiplicity equal to 1 and must be typed by a UML primitive type;
- for every class C in sW stereotyped by ≪category≫, there should exist at least an instance typed by C in isW;

Fig. 4.4 shows a possible (and valid) model representing a food ontology produced following the MOntology method.

4.4 MJavaD: Design Specification of Java Desktop Applications

Here, we present MJavaD, a modified version of the MARS design method for desktop applications based on the UML [46, 45, 47], adapted to automatize the implementation phase using a model transformation from the design model to the application source code.

The original MARS method was developed to describe how to design a JAVA desktop application, modelling such design using the UML, and provides guidelines for building a running application starting from the design. It was used in the past years in the course of Software Engineering taught in our University, were the implementation phase was carried out manually by the students. Having used this method for many years has resulted in practical test of the method itself. In other words, we have a number of designs carried out following the MARS method, and a number of the corresponding desktop JAVA applications that was developed following these designs.

As in the others precise modelling methods, even in the original MARS design method for desktop applications[10] the form of the models is determined by a conceptual metamodel. It describes the structure of a *Design Specification* that consists in different view of the designed *Application* (see [46]), each one presenting a particular aspect of it.

In MJavaD, the original structure of the *Design Specification* has been modified to remove the parts of the design specification that have no impact on the model transformation. Thus, the following views has been removed:

- *Configuration View*, that describes which are the entities composing the *Application* in some given situation. This view has been removed because it has no impact on the model transformation;
- *Additional View*, that describes how some entities part of the application interact among them to accomplish some particular task. An additional view is optional and intended just for documentation; it should not add any information not already present in the other views. This view has been removed because it has, obviously, no impact (carrying no additional information) on the model transformation.

Fig. 4.5 shows the conceptual metamodel that describes the form of UML models representing the design of a JAVA desktop application; and below we briefly describe the various parts of these models.

Data View describes all the datatypes used by the entities composing the *Application*. Technically is a UML class diagram, where all the classifiers are datatypes[11]. The datatype

[10]Indeed, MARS has been the first case of precise modelling method that has been presented.

[11]A UML datatype is a classifier whose instances are pure values, i.e., they have no identity and their state cannot be changed; thus the operations of a datatype are all queries, i.e., pure functions.

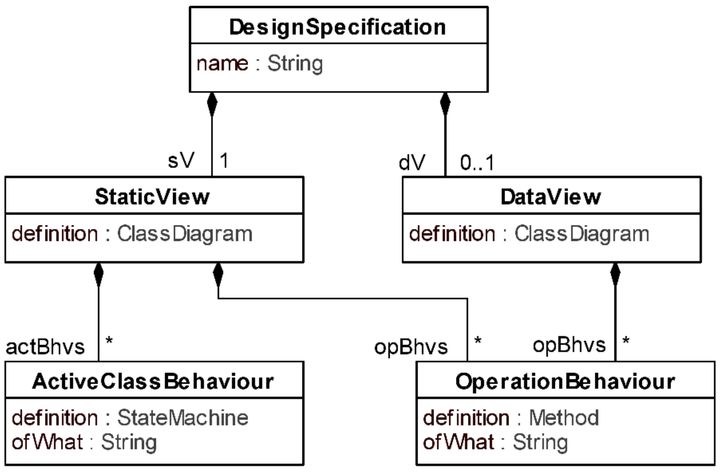


Figure 4.5: MJavaD Conceptual Metamodel

attributes must be typed by using only UML Primitive types; if a datatype is built out of other ones, then they should be linked to them by some aggregation relationships.

Static View describes all the classes typing the entities used to build the *Application*. Technically is a UML class diagram in which, where the datatypes defined in the dataview may be used to type attributes, and operation parameters. It contains classes stereotyped by ≪context≫, ≪boundary≫, ≪executor≫, ≪singletonStore≫, and ≪store≫.

Active Class Behaviour describes the behaviour of an active class that appear in the *Static View* by means of a state machine. Each active class behaviour is related with an active class (that appear in the *Static View*) by means of the *ofWhat* attribute (it is the so called context class).

Operation Behaviour describes the method associated with either an operation belonging to a passive class or to a datatype or with an operation of an active classes that is not used as call event in the state machine associated with such class. Each operation behaviour is related with an operation belonging to one of the entities that appear in the *Static View* or in the *Data View* by means of the *ofWhat* attribute.

A Java desktop application is assumed to be composed out of entities of different kinds, where each kind is modelled by a class with a specific stereotype. Below we describe all the entity/class kinds.

contex context classes represent external entities that interact with the *Application*, like humans or external system or devices, they are stereotyped by ≪context≫.

boundary classes represent entities that take care of the interaction of the *Application* with some context entities, they are stereotyped by ≪boundary≫. Boundary entities receive messages from the context entities, analyse them and afterwards either send back an immediate answer, or interact with other entities as required by the message. Boundary classes are active[12]. Thus, we describe the behaviour of boundary classes with a state machine (that it is contained in one *Active Class Behaviour* element of the *Design Specification*). Each boundary has the operations needed to define the events in the associated state machine. Since these operations are used in the state machine as events, they do not have an associated method. Each boundary has at least one operation stereotyped by ≪create≫ (along with its method to define the behaviour of the operation) that can be used to create a new instance of the class owner of the operation. Each boundary has also at least one operation stereotyped by ≪destroy≫ to end the "life" of the state machine describing its behaviour (thus, this operation must be an event of a transition to a final pseudo-state of the state machine).

executor classes represent entities that perform some core *Application* dynamic activities, they are stereotyped by ≪executor≫. They perform activities as the result of information received from the context throughout the boundaries or by other executors; during such activities they may collaborate with other entities. Also executor classes are active as boundary classes. Thus, we must describe also the behaviour of these classes by means of a state machine.

Moreover, also each executor class has operations stereotyped by ≪create≫ and ≪destroy≫ as boundary classes. Executor classes may be specialization only of abstract executor classes.

store classes represent entities that contain persistent data[13], they are stereotyped by ≪store≫. Store classes are passive[14]. Each store class has at least one operation stereotyped by ≪create≫ (along with its method to define the behaviour of the operation) that can be used to create a new instance of the class owner of the operation. Store classes may have attributes stereotyped by ≪key≫, indicating that these attributes belong to a key of persistent data represent by a store entity. Each store we may has at least one operation stereotyped by ≪destroy≫ to delete the data represented by the state of the current object. To retrieve data from a store object we may write one or more operation stereotyped by ≪finder≫ whose method must describe how data is filtered. In addition, each operation owned by a store must have a method associated with it describing its behaviour (it is contained in one *Operation Behaviour* element of the *Design Specification*).

singleton store classes represent entities that contain one single value of persistent data, they

[12]Classes can be active i.e., each of their instances is having its own thread of control.

[13]Data which are preserved also when the *Application* stops or breaks down.

[14]Classes can be passive, i.e., each of its instances executing within the context of some other object.

are stereotyped by ≪singletonStore≫. Singleton store entities are like the store entities except that they cannot own operations stereotyped by ≪destroy≫.

Obviously, a class in the *Static View* may be specialized only by a class having the same stereotype. Composition and aggregation relationships may be used and composing classes must have the same stereotype of the composed class. Associations that are not aggregation or specialization relationships are *communication associations*. A communication association from class C to C' means that an instance of class C interacts with some instances of class C'. Communication associations may connect context classes only with boundary classes, and any class in the static view must be linked by a chain of communication associations with at least a context class.

The UML Profile

The UML profile to be used following the MJavaD method is composed of the following stereotypes and tagged values (see [68, 66]):

- ≪context≫ stereotype of active class, the instances of a class stereotyped by it represent entities that interact with the *Application*.
- ≪boundary≫ stereotype of active class, the instances of a class stereotyped by it represent boundary entities.
- ≪executor≫ stereotype of active class, the instances of a class stereotyped by it represent executor entities.
- ≪store≫ stereotype of passive class, the instances of a class stereotyped by it represent store entities.
- ≪singletonStore≫ stereotype of passive class, the instances of a class stereotyped by it represent singleton store entities.
- ≪create≫ stereotype of Operation. A call to an operation stereotyped by ≪create≫ will create a new instance of the class owner of the operation.
- ≪finder≫ stereotype of Operation. Operations stereotyped ≪finder≫, may be contained only in classes that represent store or singleton store entities. A call to an operation stereotyped by ≪finder≫ may results in a sequence of objects, or in a single object of the same type of the class owner of the operation.
- ≪destroy≫ stereotype of Operation. If the stereotype of the class owner of the operation is ≪executor≫ or ≪boundary≫ a call to an operation stereotyped by ≪destroy≫ results to a transition of the state machine defining the behaviour of the class to its final state.

If the stereotype of the class owner of the operation is ≪store≫ a call to an operation stereotyped by ≪destroy≫ results in the deletion of the data represented by the state of the current object, so that these data cannot be retrieved.

- ≪key≫ stereotype of attribute. All the attributes of a class stereotyped by ≪key≫ compose the primary key of the entities represented by the class owner of these attributes. Attribute stereotyped ≪key≫ may be contained only in classes that represent store or singleton store entities.

The Well-formedness rules

Obviously, also the general UML well-formedness rules specified in Sect. 4.1 must hold on UML design models used in MJavaD.

Let *DS* be a DesignSpecification,

$SV = DS$.sV.definition be the static view,

$DV = DS$.dV.definition be the dataview,

$ACBs = DS$.sV.actBhvs be the list of the active behaviours of the active classes contained in SV,

$OBs1 = DS$.sV.opBhvs be the list of the operation behaviours of the operations owned by all the passive classes in SV, and of all the operations of the active classes in SV not used to build events in the state machiens associted with them,

$OBs2 = DS$.textsldV.opBhvs be the list of the operation behaiours of all the operations owned by datatypes in DV.

General constraints

- a class defined in SV and a datatype defined in DV must have at least one operation stereotyped by ≪create≫;
- a class defined in SV must be connected by a chain of communication associations with at least one class stereotyped by ≪context≫.
- an operation owned by a class defined in SV (except those owned by an active class without the stereotype ≪create≫ or ≪destroy≫) must be associated with a method in *OBs2*;
- each classifier and each operation must be used at least one time;

Datatype in *DV*

- an operation owned by a datatype defined in *DV*, except those stereotyped by ≪create≫, must have a return type and must be flagged as query;
- a datatype defined in *DV* must be used at least once (to type an attribute or an argument or the result of an operation);

Class defined in *SV* stereotyped by ≪context≫

- a class defined in *SV* stereotyped by ≪context≫ cannot have attributes;
- a class defined in *SV* stereotyped by ≪context≫ must be connected by a communication association at least with one class stereotyped by ≪boundary≫;
- a class defined in *SV* stereotyped by ≪context≫ may be connected with a communication association only with classes stereotyped by ≪boundary≫;

Active classes (classes defined in *SV* stereotyped by ≪boundary≫ or ≪executor≫)

- a class defined in *SV* stereotyped by ≪boundary≫ or ≪executor≫ must have an associated state machine in *ACBs* describing its behaviour;
- a class defined in *SV* stereotyped by ≪boundary≫ or ≪executor≫ must have at least one operation stereotyped by ≪destroy≫;

Class defined in *SV* stereotyped by ≪boundary≫

- a class defined in *SV* stereotyped by ≪boundary≫ may communicate only with classes stereotyped by ≪boundary≫, ≪context≫ and ≪executor≫;
- an association starting from a class defined in *SV* stereotyped by ≪boundary≫ must be named;
- an operation owned by an active class (a class stereotyped by ≪boundary≫ or ≪executor≫ defined in *SV*) that does not have the stereotype ≪create≫ or ≪destroy≫ must be used as event in the state machine in *ACBs* representing the behaviour of that class.

Class defined in *SV* stereotyped by ≪executor≫

- a class defined in *SV* stereotyped by ≪executor≫ may communicate only with classes stereotyped by ≪executor≫, ≪boundary≫, ≪store≫ and ≪singletonStore≫;
- an association between a class stereotyped ≪executor≫ and a class stereotyped ≪boundary≫ must be named;
- an association between a class stereotyped ≪executor≫ and a class stereotyped ≪store≫ or ≪singletonStore≫ must be unnamed;

Class defined in *SV* stereotyped by ≪store≫ or ≪singletonStore≫

- an operation stereotyped by ≪finder≫ must be owned by a class in *SV* stereotyped by ≪store≫ or ≪singletonStore≫;
- an operation stereotyped by ≪finder≫ must be flagged as query;
- a class in *SV* stereotyped by ≪store≫ or ≪singletonStore≫ may be related only with classes stereotyped by ≪store≫ or ≪singletonStore≫ using aggregation/composition associations;

Class defined in *SV* stereotyped by ≪store≫

- an association between a class stereotyped by ≪store≫ and a class with a stereotype ≪store≫ or ≪singletonStore≫ must be named;

State Machines

- an operation stereotyped by ≪destroy≫ owned by a class stereotyped by ≪boundary≫ must be used at least one time in the state machine in *ACBs* representing the behaviour of that class.
- a state in a state machine in *ACBs* must have a name;
- a final pseudo-state in a state machine in *ACBs* must have a name;
- a state machine in *ACBs* must have at least one transition to its final pseudo-state having as event an operation stereotyped by ≪destroy≫;

Due to space reasons, we cannot show the complete model of the design of a Java desktop application written following MJavaD. In Fig. 4.6 shows a fragment of the Static View of the design of a Java desktop application. In this figure operations signature are not showed for space reasons.

4.5 Conclusion

Here, we have presented the precise approach to build modelling methods. It prescribes that models must be written using a subset of UML and a specific UML profile; in addition they must a have a form that is precisely defined, by means of a (conceptual) metamodel and by a set of well-formdeness rules.

Moreover, in this chapter we have presented three examples of precise modelling methods starting from a very simple one (the MRelational), moving to a case of average complexity (MOntology), and finally reaching a more complex case (MJavaD).

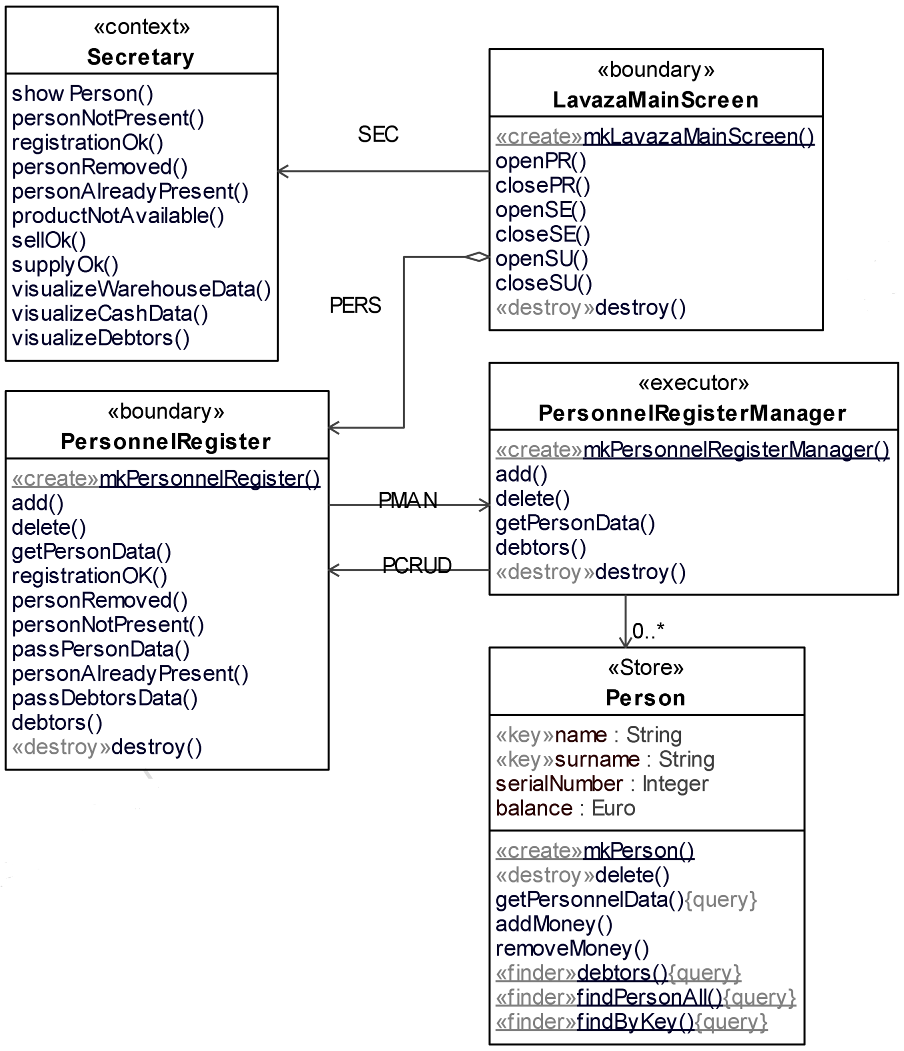


Figure 4.6: Static View Fragment

Chapter 5

MeDMoT: a Method for Developing Model to Text Transformations

In this chapter, we present a new method *MeDMoT* (*Me*thod for *D*eveloping *Mo*del to *T*ext transformations) covering all the phases of the development of model to text transformations, providing specific techniques and notations for each phase. We have pragmatically designed *MeDMoT* so that:

- it should be quite lightweight, avoiding to force the developers to use very formal notations (difficult to learn and to read), and to produce huge very detailed documentations;
- it should be able to support the development of transformations of reasonable size, not only small toy examples, e.g., the transformation from UML models into complete running Java desktop applications;
- it should be able to support the development of any kind of model to text transformations, not only from models to code written in some programming language, but, e.g., also from models to various textual artifacts required to build a modern software based system (e.g., ontology definitions using OWL, configuration files for Hibernate, HTML definitions of web pages, natural language documentations);
- it should help the developers to take any possible advantage of the available techniques, methods and tools for the target of the transformation; indeed the targets of many model to text transformations are well known and studied by long time (e.g., if the target are Java applications, then well-established architectural styles and standard testing techniques should be considered while developing a transformation from UML models);
- it should encompass all the good principles and practices of software engineering, for example the importance of high-level abstract requirements, “divide et impera” (i.e., the

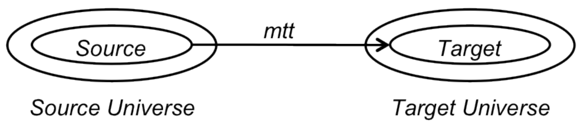


Figure 5.1: A generic MTT *mtt*

need for modularization), test before coding, etc.

MeDMoT considers model to text transformations, and prescribes how to develop a transformation guiding the developer to: (1) capture and specify the transformation requirements; (2) design the transformation; (3) implement the transformation, and (4) test the transformation.

Models that can be used as source of a transformation considered by *MeDMoT* should be produced following a precise moldering method, see Sect. 4.

In Sect. 5.1 we give an overview of the method, then in Sect. 5.2 we describe how our method define the transformation requirements. The transformation design is described in Sect. 5.3. How to test the transformation and guidelines on how to implement it are described in Sect. 5.4 and Sect. 5.5 respectively. Finally, in Sect. 5.6 we give the conclusions.

5.1 *MeDMoT* Overview

MeDMoT (**Me**thod for **D**eveloping **Mo**del **T**ransformations) aims to support the development of transformations from UML models to text, precisely it considers **M**odel to **T**ext **T**ransformations (shortly MTTs) of the kind shown in Fig. 5.1.

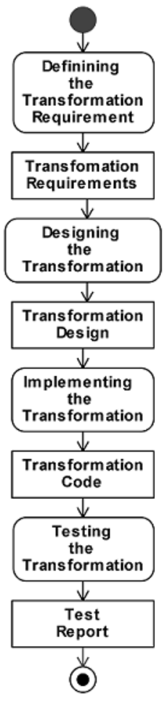
The *Source Universe* is the set of the UML models built using a specific profile, *Source* is a sub-set of *Source Universe* containing all models assumed to be correct inputs of the transformation.

The *Target Universe* is a set of **S**tructured **T**extual **A**rtifacts (shortly STAs) having a specific form, where an STA is a set of text files, written using one or more concrete syntaxes, disposed in a well-defined structure (physical positions of the files in the file system). *Target* is the subset of *Target Universe* containing all the STAs assumed to be a correct result of the transformation.

We consider STAs instead of plain text, because in many cases the relative positions of text files in the file system have a specific meaning. For example, the position of configuration files, and resource files in a Java enterprise application cannot be random but some prescribed file positions must be respected.

mtt is a function from *Source* into *Target*.

Both the elements in the source, and in the target of the considered transformation have associ-


Definining the Transformation Requirement
Transfomation Requirements
Designing the Transformation
Transformation Design
Implementing the Transformation
Transformation Code
Testing the Transformation
Test Report
Figure 5.2: *MeDMoT* activities and deliverables

ated a semantics, and the transformation developer should precisely know them. For example, if the source consists of UML models of Desktop applications and the target of Java Desktop applications, the developer must know which are the meanings of the elements composing the UML models belonging to the source, and of the Java constructs and of any API and frameworks that will be used in the target.

In the following, we describe *MeDMoT* in terms of activities to perform, deliverables to produce, and techniques to apply to obtain a STA starting from a UML model.

We can identify in *MeDMoT* the following major activities, see Fig. 5.2:

- requirement definition;
- design;
- implementation;
- testing.

Obviously, from each activity it is possible to go back at the previously stages, whenever a problem is found, but for the sake of simplicity this is not shown in Fig. 5.2.

For each activity we define guidelines driving its execution and the used notations.

To present *MeDMoT* we use U2SQL a transformation from UML models of relational data bases written using MRelational (see Sect. 4.2) to SQL DDL, as running example.

5.2 Transformation Requirements

Referring to Fig. 5.1, *MeDMoT* prescribes that a requirement specification is composed of:

- the definition of the transformation source *Source*,
- the definition of the transformation target *Target*, and
- the characterization of a relation (*R*) between source and target.

A transformation is correct with respect the above requirements iff

(s *R* $mtt(d)$ for all $s \in$ *Source*).

The relation *R* should be expressed in terms of the semantics of the transformation source and target, and not just relating metaclasses in the metamodel of the source with strings appearing in the target. For example, in the U2Java case a possible requirement is "A UML persistent class (i.e., a class stereotyped by ≪store≫) should be transformed into a Java class of the generated application made persistent by serializing it on a file", whereas in the case of U2SQL a suitable requirement is "A UML class stereotyped by ≪table≫ should be transformed into a table creation".

The suggested factorization of the requirements, that first of all leads to say explicitly what will be transformed and in what, and later to related sources and targets by considering their meanings should help the developers to decide what should do their transformation and at the same time avoiding any unnecessary over specification (e.g., fixing the details of the Java code realizing the serialization or of the SQL statement creating a table).

MeDMoT does not require to give non-functional requirements relative to the transformation, because it fixes the languages, technologies and tools to be used, and those relative to performance cannot be enforced.

5.2.1 Transformation Source

Models that can be used as source of a transformation should be models written following a Precise Modelling Method (see Sect. 4). Thus, source of a transformation is a set of UML models conform to the UML metamodel extended with a specific profile. The form of the source models is defined by a kind of conceptual metamodel, i.e., a UML class diagram with a class whose instances correspond to all the possible source models. That metamodel is constrained by a set of well-formedness rules to precisely define the set of acceptable models.

It may happen that to be able to (sensibly) define the transformation further constraints on such models have to be added.

Summarizing, the definition of the source of a transformation consists of:

1. an indication of a UML profile;
2. a conceptual metamodel of the source models, where such models are written using 1;
3. a set of well-formedness rules over 2.

U2SQL: ***Transformation Source*** Source models are defined by applying the the precise modelling method MRelational defined in Sect. 4. In this (simple) case there are not other rules to add to further restrict the set of acceptable models.

5.2.2 Transformation Target

Following the *MeDMoT*, the model transformation target is always a class of Structural Textual Artifacts with an associated semantics. A simple way to describe the target is defining its structure in terms of files and folders, and the concrete syntaxes relative to the various files composing it.

U2SQL: ***Target Definition*** In the U2SQL case, the target is a unique file containing SQL DDL statements defining a database schema. We chose to use only SQL DDL statements conform to the SQL 92 BNF[30]. We can assume that in this case the target syntax and semantics are well-known known, however we summarize them below.

A database schema consists of a set of tables. A table consists of rows (called tuples) and columns (called attributes). A table has a fixed set of columns. Each column has a defined data type (one among: VARCHAR, INTEGER, FLOAT, BOOLEAN). Each table may have at most one primary key and several (also none) foreign keys. A *primary key* for a table is a unique identifier for its rows, that is composed by a set of table columns which have a unique value for each row of the table. A *foreign key* is a referential constraint between two tables and is composed by a set of columns in the first table that reference to a primary key of the second table.

A table can be created using the SQL DDL *create* statement, but here we use a simplified version. The used concrete syntax is shown in informal way below, where the keywords are written in upper case:

CREATE TABLE *tableName* (
    columnName$_1$ *columnType*$_1$,
    . . . ,
    columnName$_n$ *columnType*$_n$
    constraint$_1$,
    . . . ,
    constraint$_k$,
)

A constraints may be either a primary key or a foreign key. A primary key can be defined as follows:

CONSTRAINT *constraintName* PRIMARY KEY(*columnName*$_1$, . . . , *columnName*$_n$)

A foreign key can be defined as follows:

CONSTRAINT *constraintName*
FOREIGN KEY (*columnName*$_1$, . . . , *columnName*$_n$)
REFERENCES *otherTableName*(*otherColumnName*$_1$, . . . , *otherColumnName*$_n$)

where:

- *otherTableName* is the table referenced by the foreign key;
- *otherColumnName*$_1$, . . . , *otherColumnName*$_n$ are columns of *otherTableName* that identify a single row of that table;
- *columnName*$_1$, . . . , *columnName*$_n$ are the columns of the table been defined whose values identify the row in the other table;

Only one primary key constraint for table is permitted.

5.2.3 Characterization of Transformation Relation

MeDMoT guidelines for capturing the requirements suggest to try to characterize the transformation relation *R* referring to the semantics of *Source* and *Target* elements and taking advantage of the existing techniques for expressing requirements or just properties on the target.

R may be defined in the following way:

$R = \{(a,b) \in \textsf{\textit{Source}} \times \textsf{\textit{Target}} \mid b \text{ satisfies } Reqs(a)\}$

where $Reqs$: $\textsf{\textit{Source}} \rightarrow (\textsf{\textit{Target}} \rightarrow \{true, false\})$, i.e., $Reqs$ given an input model returns a set of "usual/standard" requirements/properties/conditions on *Target*.

The available knowhow on the target will help to decide which kind of properties to consider and how to express them. For example, if *Target* is the set of Java interactive programs we can use mandatory and prohibited execution scenarios, if *Target* is instead the set of the SQL scripts for creating a database, we can check if the created database has/has not a table with certain properties.

With this approach the requirements on the transformations may be defined at a (good) high level of abstraction, because only the semantics and the properties of the outputs are considered and not their specific syntactic form (remind that almost in any case there are many different elements in the target that have the same semantics, and the requirements should not distinguish among them).

U2SQL: Characterization of Transformation Relation

In the U2SQL case we characterize the transformation relation relation between source and target elements as follows:

A class with the stereotype ≪table≫ corresponds to a table with the same name and each class attribute corresponds to a column of the table. The attribute types are mapped in the SQL types as described in Table 5.1.

All the attributes of a class with the stereotype ≪key≫ correspond to a primary key (of the table corresponding to that class) composed by the columns corresponding to the class attributes stereotyped ≪key≫.

An oriented association from a class *Ca* to a class *Cb*, both stereotyped ≪table≫, corresponds to a foreign key definition in the table *Cb*.

UML	**SQL**
Integer	INTEGER
UnlimitedNatural	INTEGER
String	VARCHAR
Real	FLOAT

Table 5.1: U2SQL Types transformation

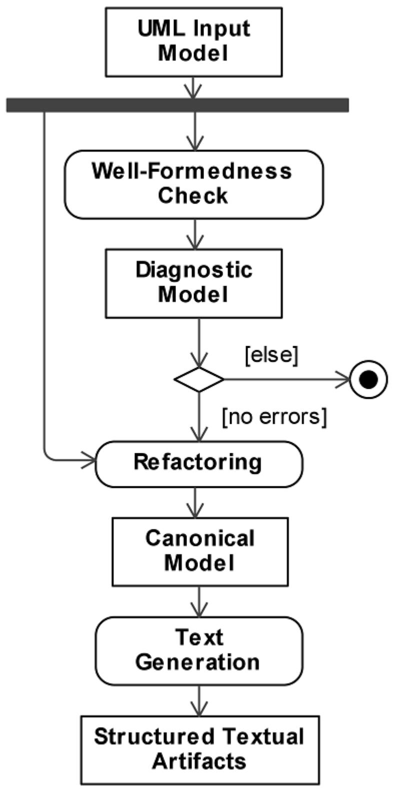


Figure 5.3: Model to Text Transformation architecture

5.3 Model Transformation Design

MeDMoT prescribes a particular architecture, shown in Fig. 5.3, for the developed MTTs, structured as a chain of transformations of different types, some from model to model and one from model to text. This specific architecture will help to modularize the transformation and to decompose the work of the developers in smaller tasks.

Well-Formedness Check The input model is verified by a model to model transformation producing a diagnostic model to see if it satisfies the well-formedness rules that constrain the transformation source (see Sect. 5.2.1).

Refactoring If the input model is well-formed, then it is simplified by one or more Model to Model Transformations performed in a well-defined order. The refactoring has been introduced to facilitate the following task.

Text Generation Finally, the simplified model is transformed into an STA using a MTT.

Obviously the designed transformation should take as input a model *i* and generate an STA *o* such that *i R o*, where *R* is the transformation relation part of the requirement specification, described in Sect. 5.2.3.

5.3.1 Well-Formedness Check

The input model must be checked to verify if it is an acceptable source of the transformation as defined in the requirement phase. Following *MeDMoT*, during the activity of the definition of the source conceptual metamodel, we had to write a set of well-formedness rules, see Sect. 4.1.

This set of rules may contain some rules that are not easily computable, or just non-computable (e.g., Moreover, there are some cases in which we want to further constraint the form of the source models for the purpose of simplify the following transformations (or just make them possible). In these cases we have to write some other rules that we call implementation dependant. Thus, starting from the original set of rules we derive a new rule set by adding implementation dependent rules and removing all those that are not computable (or are just difficult to compute).

As suggested in [53] the verification can be considered as a function *f* returning a *diagnostic model* starting from the input model, *f: Model → DiagModel.*

In *MeDMoT* the function *f* is a Model to Model transformation that starting from the input model produces a *diagnostic model.*

The diagnostic model, result of the well-formedness check, must keep the information about the violations of well-formedness rules over the input model. The diagnostic model, as any other model, needs a metamodel to specify its form. We call it the *Diagnostic Metamodel.* It must specify error concepts, one for the violation of each of the well-formedness rules. The Diagnostic metamodel must contain an abstract class named Error, such that each class specializing it will represent the violation of a specific well-formedness rule. It is also a useful way to visually summarize all the possible errors.

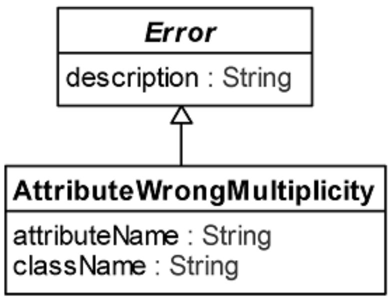


Figure 5.4: Example of diagnostic metamodel

Moreover, representing the result of the rules evaluation as a model give us the possibility to transform it in other forms, such as text, xml or natural language, to build a human-readable error report, or to compute metrics such as the number of errors of one specific kind.

Fig. 5.4 shows an example of diagnostic metamodel. In this case there is only one type of error elements, indeed only a class AttributeWrongMultiplicity specializes Error, whose elements contain a description of the error occurrence, the name of the attribute with the wrong multiplicity and the name of class owner of that attribute. If the transformation implementing the check function is designed carefully, the description attribute may contain a natural language description of the violation which includes an indications of the model elements that violate the constraint.

U2SQL: Well-Formedness Check Fig. 5.5 shows the Diagnostic metamodel for the U2SQL case, that allows to represent all the violations of the well-formedness rules listed in Sect. 4.2.

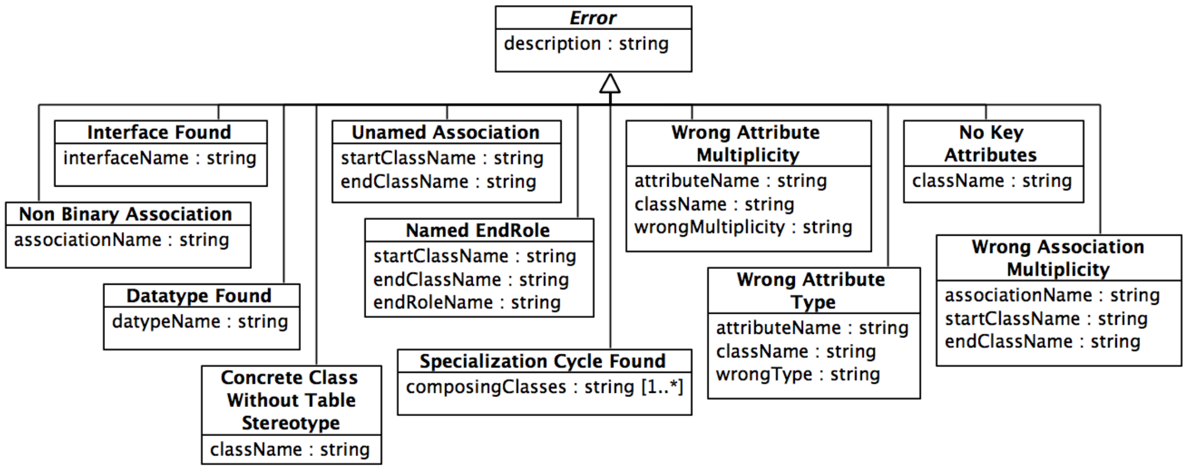


Figure 5.5: U2SQL Diagnostic Metamodel

5.3.2 Refactoring

As shown in Fig. 5.3, after having verified the well-formedness of the input model, it must be transformed into an STA. The first step in transforming the input model into an STA is the *refactoring*.

Refactoring is a well-known technique to change the internal structure of software to improve certain characteristics [88], and is a typical example of endogenous model transformations (i.e., transformations between models expressed in the same language) [89].

The main reason to apply some refactorings to the source models, before transforming them into STAs, is to reduce the complexity of the source model, and consequently the complexity of the final transformation from models to text.

In general, many features and constructs of the source models may be semantically equivalent to combinations of other constructs, in this case the refactoring will transform them into the equivalent ones, so the following transformation to text will have to cope with a reduced set of features/constructs.

For example, in the case of U2SQL, a specialization relationship between two classes (say, A specializes B) means that A has also all the attributes and all the associations of B; thus it can be removed if we add to A all the attributes and associations of B. Thus, the specialization relationships may be removed from the source models, and the subsequent transformation into SQL code does not need to worry about how to realize specialization in a relational database.

Another reason to use refactoring is that the production of the input model can be simplified (less time required to produce a semantically equivalent model) introducing some shortcuts, that, by means of refactoring, can be eliminated, and so their presence does not affect the production of the STA. For example, if in the input model there is a tree hierarchy of classes and all the classes must be stereotyped with the same stereotype, then a convention may allow to use the stereotype only in the root class, the refactoring process will apply the same stereotype to all the classes of the hierarchy, before the model is transformed into STA.

The various refactorings transformations must be executed in a well-defined order.

MeDMoT suggests that "Each refactoring transformation should change only one type of elements of the input model (or, more generally, to perform a refactoring as simple as possible)".

We present the design of the refactoring transformations *by examples*, and this technique will be used also for presenting the design of the subsequent MTT for text generation. *MeDMoT* requires to give for each transformation an informal description and a set of examples of application, that we call *transformation case* or just cases. Each "*transformation case*" presents a generic example of application of the transformation. Technically, it is a pair, where the first component is an input model template, and the second is the result of the transformation applied on it. The terms "*transformation case*" were first used in [70], from which we have taken some inspiration. They use *transformation cases* only in the analysis phase of a model to model transformation, to describe how concrete source models are transformed into the target ones.

The design of a refactoring transformation is then presented in the following way:

- Refactoring name

refactoring informal description

Refactoring case

$\text{source}_1 \longrightarrow \text{target}_1$

....

Refactoring case

$\text{source}_n \longrightarrow \text{target}_n$

where source_i, target_i $(i = 1, \ldots, n)$ are templates for source models, i.e., source models where variables may appear; obviously such variables are implicitly typed by model elements, e.g., class name, list of attributes, and association end multiplicity.

Fig. 5.6 shows an example of a template for a class, where "Cname", "attrs" and"ops" are three variables typed respectively by Class Name, list of attributes and list of operations.

Moreover, it is possible to add some conditions on the variables appearing in the source templates (obviously, this means that they maybe instantiated only by values satisfying such conditions).

MeDMoT provides the following guidelines to help to design a transformation by examples:

- starting from the simplest template (even composed by a single parameter), write the right side of the transformation case. Then, if the parameter/s in the template are not enough to complete the right side, start to refine the template exposing its structure so that the needed parameter/s will appear, until the right side of the transformation case can be completed; it may happen that you discover that there is the need to introduce many different templates (e.g., the transformation of a class with attributes and no operations is totally different from the transformation of a class with operations and no attributes).
- try to ensure that there are enough examples (transformation cases) to allow to specify how to transform all the aspects of the source models relevant for the transformation itself;

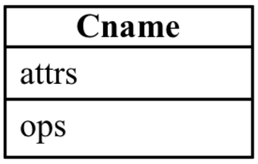
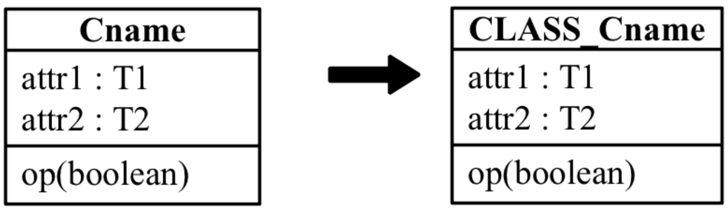
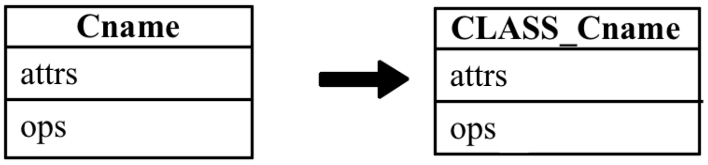


Figure 5.6: Example of Model Template



(a) Unnecessary details



(b) Correct level of details

Figure 5.7: Source Templates

- eliminate the cases whose source may be obtained by instantiating the variables appearing in the source of another case;
- avoid unnecessary details in the sources, i.e., try to have the most abstract sources (always check if a source fragment may be replaced by a variable, for example the source of the transformation case shown in Fig. 5.7(a) is unnecessarily detailed, a better way to present it is instead the one shown in Fig. 5.7(b).

The proposed way of giving the design is easy to understand, e.g., it does not require an in-depth knowledge of the source metamodel; on the other hand it is critical that the chosen transformation cases are meaningful and sufficient to specify how to implement the transformation.

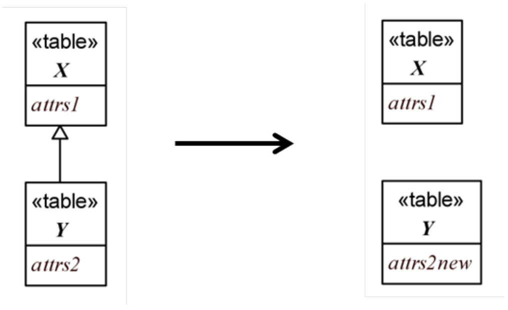
U2SQL*: *Refactoring

In the U2SQL case we decided to apply two refactorings: (1) remove specialization relations and (2) remove many to many associations.

- Remove specializations

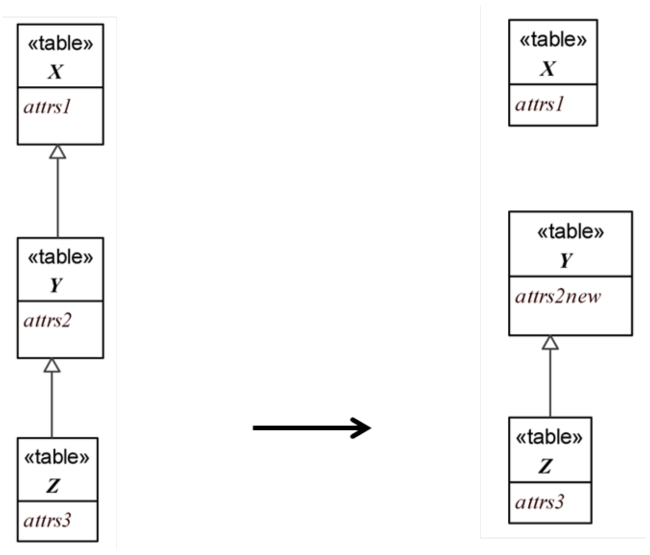
Any specialization relation is removed by duplicating attributes and associations of the superclasses in the subclasses. Abstract classes are then removed.

Refactoring case



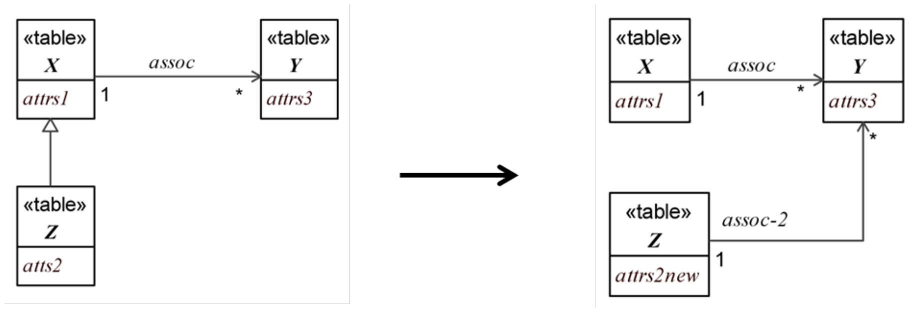
where:
– the class *X* of the template must be the root of the inheritance tree.
– *attrs2new* is the union of the attributes belonging the the class *X*, *attrs1*, and the attributes of the class *Y*, *attrs2*. Attributes with the same name and type are taken only once.

Refactoring case



where:
– the class *X* of the template must be the root of the inheritance tree.
– *attrs2new* is the union of the attributes belonging the the class *X*, *attrs1*, and the attributes of the class *Y*, *attrs2*. Attributes with the same name and type are taken only once.

Refactoring case

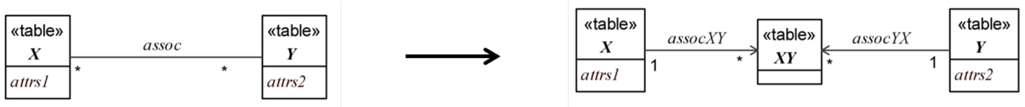


where:
– the class *X* of the template must be the root of the inheritance tree.
– *attrs2new* is the union of the attributes belonging the the class *X*, *attrs1*, and the attributes of the class *Z*, *attrs2*. Attributes with the same name and type are taken only once.

- Removal of Many-to-Many Associations

A many-to-many association is replaced by a new class and two one-to-many associations.

Refactoring case



5.3.3 Design of Text Generation Transformation

MeDMoT proposes to proceed in the following way to design the final component of a MTT, see Fig. 5.3, i.e., the **T**ext **G**eneration **T**ransformation (shortly TGT) that is a model to text transformation. First, for each possible input the output of the transformation must be designed, taking advantage of any existing techniques and methods, and trying to meet the requirements. Then, the TGT must be designed, and the design must be showed using an appropriate notation. The following sections describe these steps.

Design of the Transformation Output

The output of the transformation must be an element of the target as described in Sect. 5.2.2.

The developer needs to "design" the output of the transformation for each possible input model, or to be more precisely to have in mind such design, obviously taking into account all the requirements expressed before: for each input model im, the designed output of the transformation, say o should be such that $im\ R\ o$ (R defined in Sect. 5.2.3). Following a model driven approach for a software engineering task means essentially, in our opinion, to uplift an already known activity from the single case to a generic case represented by a model, but however the developer must know how to perform such activity.

Thus, the transformation developer should, as first step, *design* the target element result of the transformation of each input model; all methods, techniques, know-how relative to the target should be used at this point. For example, in the U2Java case everything relative to design Java applications should be considered, e.g., to use a three-tiered architecture or taking advantage of the Hibernate framework.

U2SQL: Design of the Transformation Output In this simple case we have to write SQL DDL statements to create one or more tables containing an arbitrary number of columns. Moreover, tables must have a primary key composed by one or more columns and one or more foreign keys referencing other tables. Thus, we have substantially two choices: (a) use the *CREATE TABLE* statement containing all the other needed statements; (b) use the *CREATE TABLE* statement and add all the other needed table features using appropriates *ALTER TABLE* statements.

We have decided to write SQL DDL statements following the (a) option. Thus, each CREATE TABLE statements contains all the SQL DDL statements that describe all the table features.

Text Generation Transformation Architecture

Since the TGT (i.e., the last step of the whole MTT, see Fig. 5.3) may be quite large and complex, it should be designed as composed by many sub-transformations, each of them taking care to transform parts of the input models. The various sub-transformations may be arranged in a kind of call-tree (we use a diagram similar to the structure chart[1], that we call functional *decomposition diagram*), where the nodes are labelled by the sub-transformations themselves and the children of a node labelled by S are the trees corresponding to those called by S.

An example of decomposition diagram is shown in Fig. 5.8

U2SQL: Decomposition Diagram

In Fig. 5.8 is shown the decomposition diagram relative to the U2SQL case.

Details and description of each transformation function are showed in the TGT Design.

[1]see `http://en.wikipedia.org/wiki/Structure_chart`

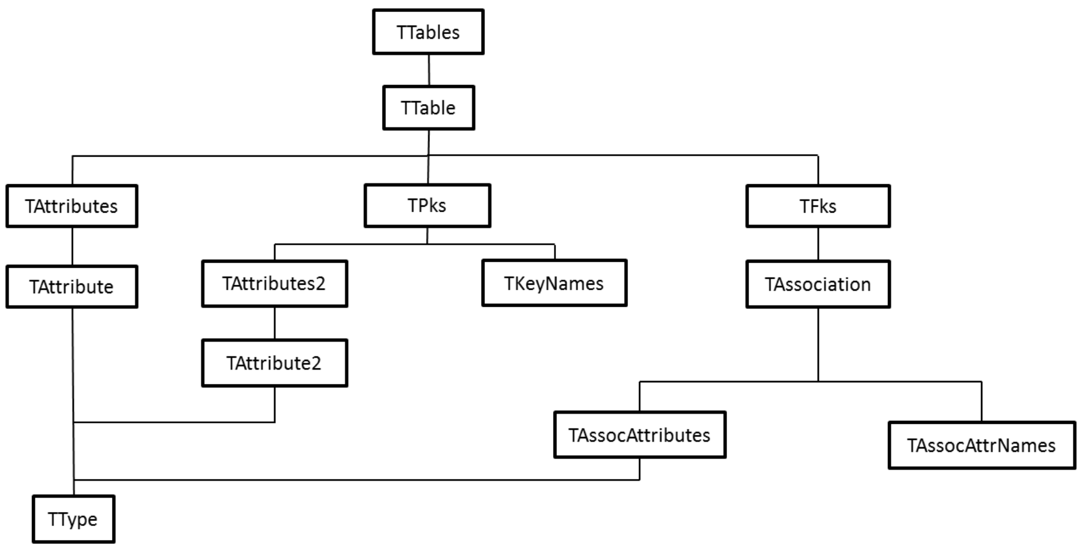


Figure 5.8: U2SQL: Decomposition Diagram

Text Generation Transformation Design

As said before the TGT is expressed as the composition of a set of transformation functions, and the design of each function is presented again by examples or better by transformation cases, as made before for the refactoring transformations in Sect. 5.3.2.

Whenever needed, the design of the TGT proceed from a high to a lower level of abstraction by means of a process of refinement. We call the design at high level of abstraction *User Level Design* and the one at lower level of abstraction *Developer Level Design*.

User Level Design The *User Level Design* is done referring at the same conceptual metamodel of the source models used in the requirement definition. This metamodel has a high level of abstraction (it is based on UML diagrams) and does not require a deep knowledge about the UML metamodel. For this reason the *User Level Design* does not require any further activity before the design can start.

Developer Level Design The *Developer Level Design* is done referring to a more detailed conceptual metamodel. This metamodel is a kind of "simplified" UML metamodel taking in consideration only those metamodel elements needed to describe the transformation source. Therefore, before this kind of design may start a further effort is required. Indeed, the new metamodel must be developed.

Text Generation Transformation Design -*User Level Design*-

The design of transformation function is done following the same ideas and notations used in the refactoring design. Thus, in this case the design of a transformation function in given in the following way:

- **Transform**(ParameterType$_1$, . . . , ParameterType$_n$)

Informal description

Transformation case

---

templ$_1^1$ . . . templ$_n^1$ $\longrightarrow$ text$_1$

. . .

Transformation case

---

templ$_1^r$ . . . templ$_n^r$ $\longrightarrow$ text$_r$

Each transformation case presents an example of use of the considered function. These examples of use are pairs shown using the concrete syntax of the source and target respectively. On the left of the arrow there are templates for the parameters, and on the right there is the result of the function applied to such templates. As for the refactoring transformations, the left side may contain conditions on the variables appearing in the templates. In certain cases, the right side may contain, in addition to the concrete syntaxes used in the target, graphical symbols representing a folder or a file, since in the STA the relative position in the file system of the produced textual artifacts plays an important role. Moreover, the right side may contain calls to other functions.

Whenever it is possible a function may be simply presented by its definition, as follows:

- **Transform**(ParameterType$_1$, . . . , ParameterType$_n$)

Informal description

Definition

Transform(Templ$_1$, . . . , Templ$_n$) = text

In some case, may be useful to describe a particular type of functions, which do not produce any kind of output, but simply query the source model to see if it contains a specific kind of element/s. The design of these functions, that we call *queries*, is done in the following way:

- **Query**(ParameterType$_1$, …, ParameterType$_n$)

Informal description

Transformation case

---

templ$_1^1$ … templ$_n^1$ $\longrightarrow$ TRUE/FALSE

…

Transformation case

---

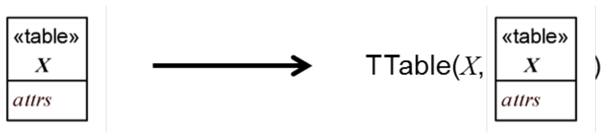
templ$_1^r$ … templ$_n^r$ $\longrightarrow$ TRUE/FALSE

U2SQL**:** ***Text Generation Transformation Design -User Level Design-*** The design takes into consideration the transformation of UML model already refactored (see the example part of Sect. 5.3.2). Thus, these UML input models do not contain specialization relations, nor many-to-many associations. In the following we describe all the functions contained in the decomposition diagram.

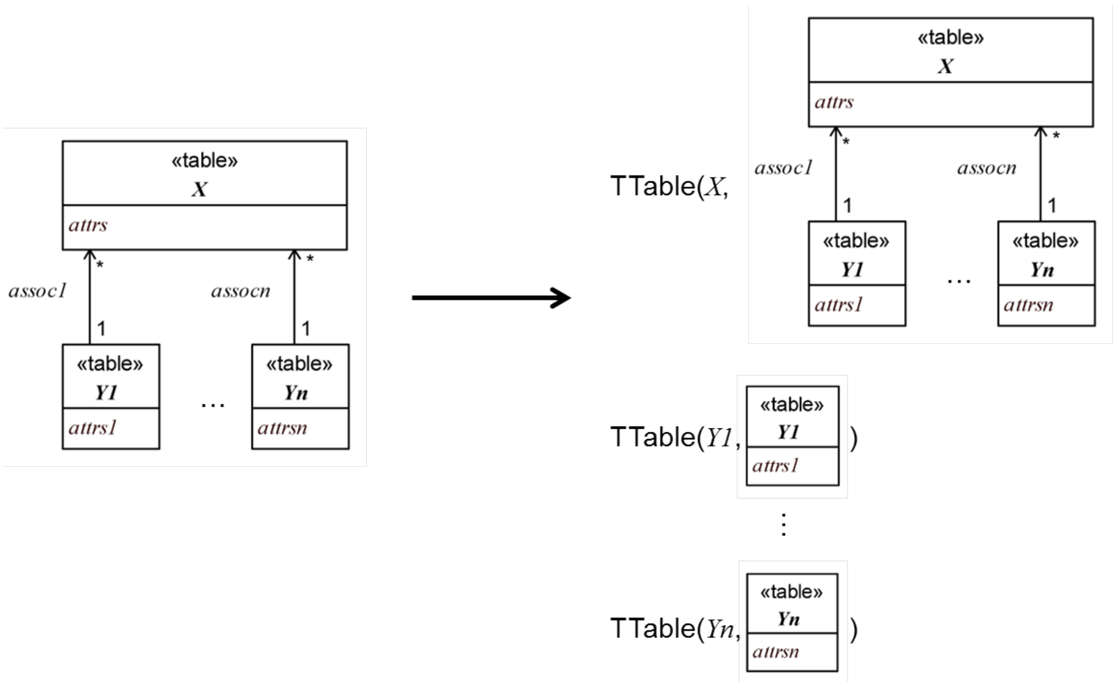
- **TTables**(ClassDiagram)

It transforms a Class Diagram containing tables into the corresponding SQL-DDL statements creating those tables.

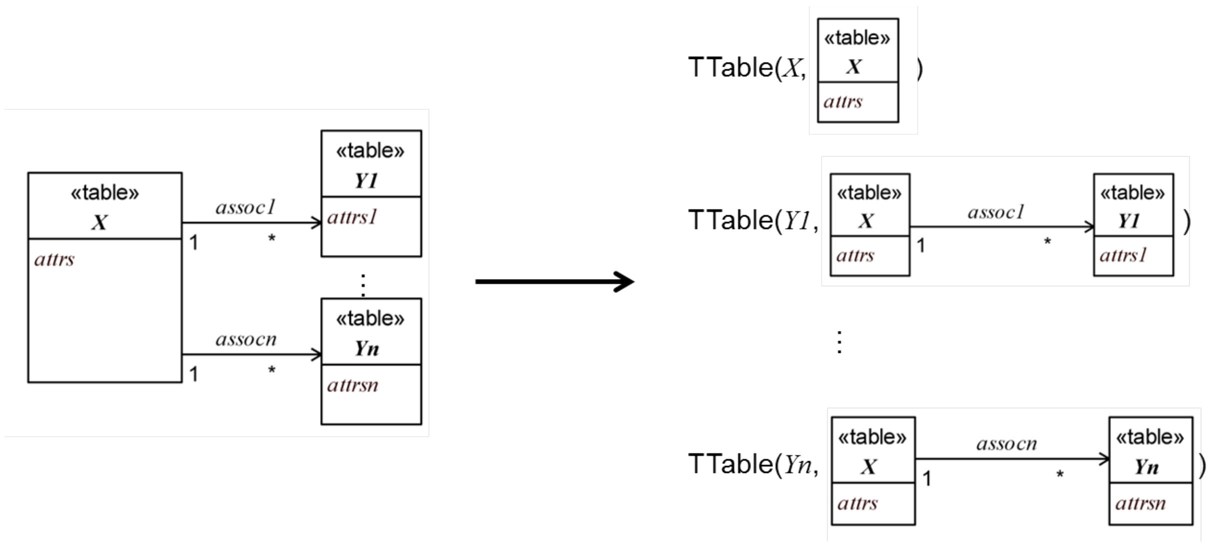
Transformation case



Transformation case



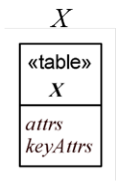
Transformation case


«table»
X
attrs
assoc1
1
*
«table»
Y1
attrs1
assocn
1
*
«table»
Yn
attrsn
TTable(X, «table» X attrs)
TTable(Y1, «table» X attrs assoc1 1 * «table» Y1 attrs1)
TTable(Yn, «table» X attrs assocn 1 * «table» Yn attrsn)

- **TTable**(String,ClassDiagram)

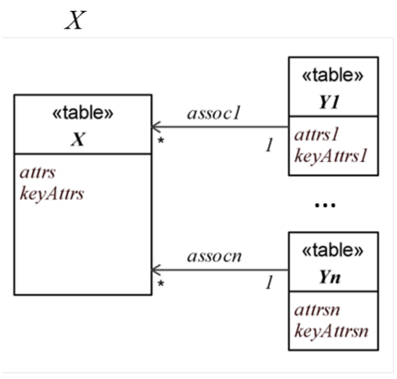
It transforms a class stereotyped by ≪table≫ into the SQL-DDL statements that represent the creation of that table (the parameters are the table name and the model fragment containing it, respectively). There are two major cases: (1) the table has at least one incoming direct association and (2) the table is without relation with other tables.

Transformation case



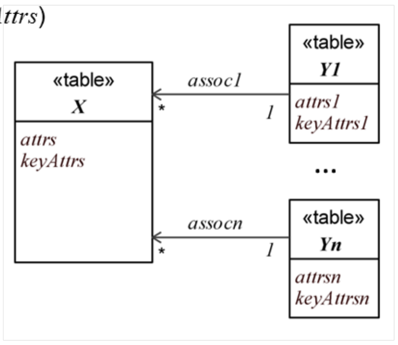
```
CREATE  TABLE  X(
  TAttributes(attrs)
  TPk(X,keyAttrs)
);
```

Transformation case



CREATE TABLE X (
TAttributes(*attrs*)
TPk(X,*keyAttrs*)

TFks(X,



)

);

- **TAttributes**(Sequence(Attribute))

It transforms all the class attributes into the corresponding column definitions.

Definition

TAttributes($attr_1, \ldots, attr_n$) = **TAttribute**($attr_1$), ..., **TAttribute**($attr_n$)

- **TAttribute**(Attribute)

It transforms a class attribute into the corresponding column definition.

Definition

TAttribute(*attr*:*type*) = *attr* **TType**(*type*)

- **TPKs**(String,Sequence(Attribute))

It transforms a sequence of class attributes stereotyped by ≪key≫ into a primary key definition using the columns corresponding to these attributes.

Definition

TPKs(C, $keyAttr_1, \ldots, keyAttr_n$) =
TAttributes2($keyAttr_1, \ldots, keyAttr_n$)
CONSTRAINT *C*Pk
PRIMARY KEY(**TKeyNames**($keyAttr_1, \ldots, keyAttr_n$))

• **TKeyNames**(Sequence(Attribute))

It transforms a sequence of class attributes stereotyped by ≪key≫ into the sequence made by their names.

Definition

TKeyNames(key_1:$type_1$ … key_n:$type_n$) = $key_1, \dots, key_n$

• **TAttributes2**(Sequence(Attribute))

It transforms all the class attributes stereotyped ≪key≫ into the corresponding column definitions.

Definition

TAttributes2($KeyAttr_1, \dots, KeyAttr_n$) =
TAttribute2($KeyAttr_1$), …, **TAttribute2**($KeyAttr_n$))

• **TAttribute2**(Attribute)

It transforms a class attribute into the corresponding column definition.

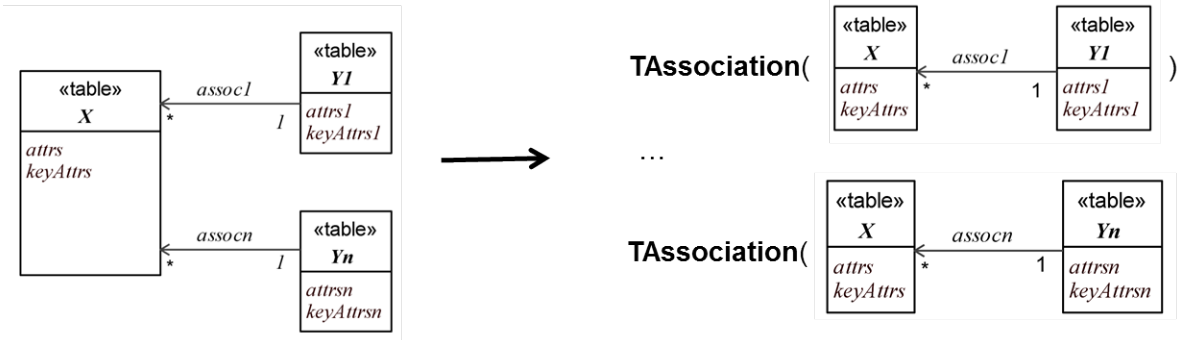
Definition

TAttribute2(*attr*:*type*) = *attr***TType**(*type*) NOT NULL

• **TFKs**(ClassDiagram)

It transforms a sequence of associations incoming in a class into the corresponding foreign key definitions.

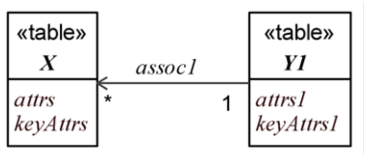
Transformation case



- **TAssociation**(ClassDiagram)

It transforms an incoming directed association into a foreign key definition.

Transformation case



$\longrightarrow$

TAssocAttributes(*assoc1*,*keyAttrs1*)
CONSTRAINTS *assoc1*FK
FOREIGN KEY(**TAssocAttrNames**(*assoc1*,*KeyAttrs1*))
REFERENCES *X*(**TKeyNames**(*keyAttrs1*)

- **TAssocAttributes**(String, Sequence(Attribute))

It transforms the name of an incoming association and the name of the attributes stereotyped by ≪key≫ owned by the class from which this association start into the set of the column definition used to store the values of the foreign key.

Definition

TAssocAttributes(*assocIn*,$attr_1$:$type_1$, . . . , $attr_n$:$type_n$) =
assocIn-$attr_1$ **TType**($type_1$),
. . . ,
assocIn-$attr_n$ **TType**($type_n$)

- **TAssocAttrNames**(String, Sequence(Attribute))

It transforms the name of an incoming association and the names of the attributes stereotyped by ≪key≫ owned by the class from which this association start into the set of names of the columns used to store the values of the foreign key.

Definition

TAssocAttrNames($assocIn_1$, $attr_1$:$type_1$, . . . , $attr_n$:$type_n$) = *assocIn*-$attr_1$, . . . , *assocIn*-$attr_n$

Text Generation Transformation Design Notation -*Developer Level Design*-

The model to text transformation design at the developer level is expressed as a conceptual meta-model and a set of transformation functions. The definition of each function is presented following the schema shown below.

- **Transform**(ParameterType_1, . . . ,ParameterType_n)

Informal description

Definition

Transform$(X_1, \ldots, X_n)$ = text

The parameter types are either defined in the conceptual metamodel or are OCL types. In the right side calls to other functions may appear. These calls can use the actual parameters and selectors referring the conceptual metamodel.

The source conceptual metamodel used in the *Developer Level Design* is a kind of simplified UML metamodel. This metamodel can be described as follows:

Metamodel Elements Elements composing the meta-model are a representation of the UML profiled metamodel. This metamodel contains only elements needed to represent models of the transformation source. Profiled UML metamodel elements are represented as a single element (extension relation and UML meta-classes are not represented).

Relations Among Meta-Model Elements The relations among the metamodel elements are those natural of the source being represented.

Metamodel elements must be described with a short sentence and the UML meta-class that is in relation with the element being described.

U2SQL: Source Conceptual Metamodel and Text Generation Transformation Design - Developer Level Design-

Fig. 5.9 shows the conceptual metamodel of the source and the Table 5.2 shows the description of this metamodel.

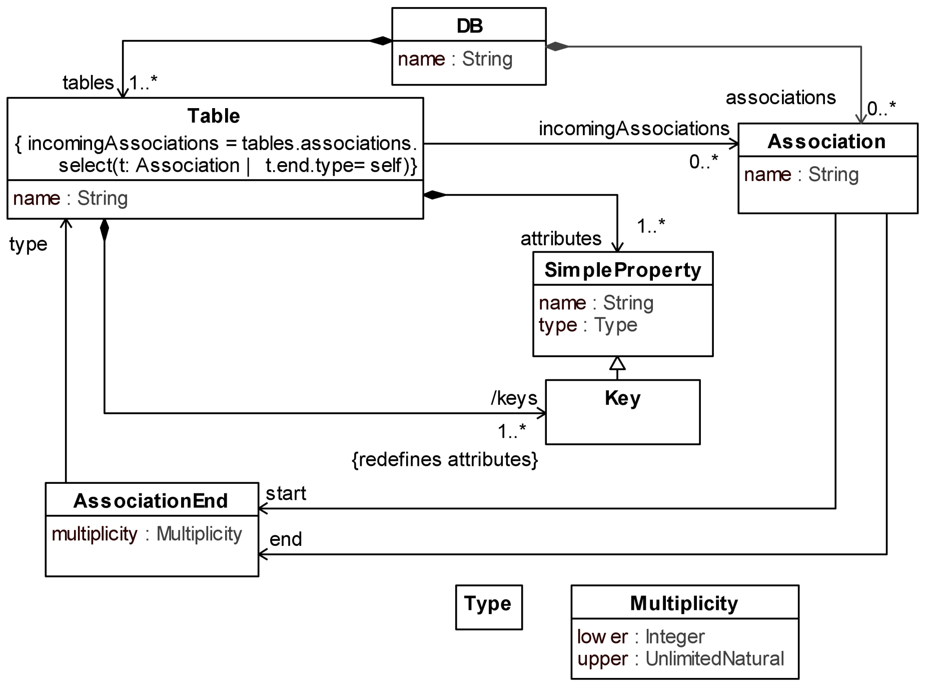


Figure 5.9: U2SQL Source Conceptual Metamodel *Developer Level Design*

Element Name	**Description**	**Related UML Metamodel Element**
DB	A SQL schema definition	UML::Model
Table	A class stereotyped by ≪table≫	UML::Class
SimpleProperty	An attribute of a class	UML::Property
Key	An attribute of a class stereotyped by ≪key≫	UML::Property
AssociationEnd	An association end, with its name and multiplicity	UML::Property
Multiplicity	A multiplicity used to define association cardinalities	UML::MultiplicityElement
Type	A type allowed for attributes	UML::PrimitiveTypes

Table 5.2: U2SQL: Source Conceptual Metamodel *Developer Level Design* Element Description

- **TTables**(Sequence(DB.tables)))

It transforms a sequence of class stereotyped ≪table≫ into the corresponding SQL-DDL statements creating these tables.
TTables($Table_1, \dots, Table_n$) = **TTable**($Table_1$) ... **TTable**($Table_n$)

- **TTable**(Table)

It transforms a table into the corresponding SQL-DDL statements creating this table.
TTable(*Table*) = CREATE TABLE *Table*.name (
TAttributes(*Table*.attributes),
TPKs(*Table*),
TFKs(*Table*.incomingAssociations)
)

- **TAttributes**(Sequence(Attribute))

It transforms all the class attributes into the corresponding column definitions.
TAttributes($attr_1, \dots, attr_n$) = **TAttribute**($attr_1$) ... **TAttribute**($attr_n$)

- **TAttribute**(Attribute)

It transforms a class attribute into the corresponding column definition.
TAttribute(*attr*) = *attr*.name **TType**(*attr*.type)

- **TPKs**(Table)

It transforms a sequence of class attributes stereotyped by ≪key≫ into a primary key definition using the columns corresponding to these attributes.

TPKs(*Table*) = CONSTRAINT *Table*.name+"PK"
TAttributes2(*Table*.keys)
PRIMARY KEY(**TKeyNames**(*Table*.keys))

- **TKeyNames**(Sequence(Key))

It transforms a sequence of class attributes stereotyped by ≪key≫ into a sequence of names of the attributes themselves.
TKeyNames($key_1 \dots key_n$) = key_1.name, ..., key_n.name

- **TAttributes2**(Sequence(Attribute))

It transforms the class attributes with the ≪key≫ stereotype into the corresponding column definitions.
TAttributes($attr_1, \dots, attr_n$) = **TAttribute2**($attr_1$) ... **TAttribute2**($attr_n$)

- **TAttribute2**(Attribute)

It transforms a class attribute with the ≪key≫ stereotype into the corresponding column definition.
TAttribute() = *attr*.name **TType**(*attr*.type)

- **TFKs**(Sequence(Association))

It transforms a sequence of associations incoming in a class into the corresponding foreign key definition.
TFKs($ssoc_1, \ldots, assoc_n$) = **TAssociation**($assoc_1$) … **TAssociation**($assoc_n$)

- **TAssociation**(Association)

It transforms an incoming directed association into a foreign key definition.
TAssociation(*assoc*) =

```
if assoc.end.multiplicity.upper = * then
                    TAssocAttributes(assoc.name.end.type.keys)
                    CONSTRAINTS assoc.name+"FK"
                    FOREIGN     KEY(TAssocAttrNames(assoc.name,assoc.start.type.keys))
                    REFERENCES assoc.start.type.name(TKeyNames(assoc.start.type)) else
""
```

- **TAssocAttrNames**(String, Sequence(attribute))

It transforms the name of an incoming association and the names of the attributes stereotyped by ≪key≫ owned by the class from which this association start into the set of names of the columns used to store the values of the foreign key.
TAssocAttrNames(*assocName*, $attr_1, \ldots, attr_n$) =
    assocName+"-"+ *attr*.name … *assocName*+"-"+ $attr_n$.name

- **TAssocAttributes**(String, Sequence(attribute))

It transforms the name of an incoming association and the name of the attributes stereotyped by ≪key≫ owned by the class from which this association start into the set of the column definitions used to store the values of the foreign key.
TAssocAttributes(*assocName*, $attr_1, \ldots, attr_n$) =
assocName+"-"+ $attr_1$.name **TType**($attr_1$.type),
…,
assocName+"-"+ $attr_n$.name **TType**($attr_n$.type)

5.4 Model Transformation Testing

Like any other piece of software, model transformations must be designed, implemented and tested. Testing model transformations is a problem more complex than code testing [50]. Indeed, there are several factors to be considered when evaluating the complexity of this problem:

- there are a lot of model transformation languages, some are general purpose languages (e.g., Java or C++), others are designed for specific tasks such as model to model (M2M) transformations (e.g., ATL, QVT, Kermeta [64]) and model to code (M2C) translations (e.g.,Acceleo, Jet, XPand), the latter can be considered as a special case of model-to-text transformation language. This heterogeneity must be taken into account especially in the selection (or definition) of white-box testing technique [51];
- selecting input for model transformations is more difficult than selecting input for programs testing because they are more difficult to be defined in an effective way (e.g., compare defining a set of input values for a program with defining a whole UML model). Hence, also defining adequacy criteria for MTTs is again more difficult than in the case of programs;
- the definition of oracle functions for model transformations is difficult due to the complex nature of models. When talking about oracle functions for model transformation tests we must analyse the validity of produced model; this require to analyse syntactic and semantic properties of the output [91];
- writing test cases, managing input and output models, metamodels and model transformation programs require a set of support tools, especially in the case of a chain of transformations, where various model transformation languages may be used. All these tools should be integrated as much as possible, minimizing interoperability problems;
- to test model transformations we need a specification of what the transformation has to do; in general writing specification for model transformations is not easy, and often they are described only in very informal terms;
- input and output of model transformations are models, that are complex data structure [52]. They are often large and require the use of (sometimes complex) tools for their production.
- defining adequacy criteria for MTTs is again more difficult than in the case of programs.

Currently there are no standards or well-established proposals available for transformation testing, especially in the case of model-to-code transformations and transformation chains. Moreover, we seek for model transformation testing approaches which may scale up to “real” cases;

For these reasons, we try to define a method for testing MTTs, whose application is simple and that can scale with the size of the tested model transformations. Moreover, our approach to MTTs testing is pragmatic and has its main purpose in providing guidelines to design and implement tests for them taking into account the extreme variability of their targets.

Part of the content of this section has been published in our works [106] and [107].

In this section we give a transformation test classification Sect. 5.4.1 in which we define three kind of tests that can be made over model to text transformations, then we show how to do a kind of unit testing Sect. 5.4.2. After that, in the Sect. 5.4.3 we define three testing adequacy criteria to build appropriate testing models. To complete our the description of model transformation testing, we describe in Sect. 5.4.4 how testing automation can be achieved and finally we give a description of our method to test model to text transformation Sect. 5.4.5. Regression test and a way to use the these techniques to test input models are described respectively in Sect. 5.4.6 and in Sect. 5.4.7.

5.4.1 Transformation Test Classification

A transformation *test case* is a triple $(IM, check, expct)$, where $IM \in \mathsf{Source}$ is an input model, *check*: $\mathsf{Target} \to CheckResult$ is a total function, and $expct \in CheckResult$. *check* represents some observations on the output models, where the result of such observations will be some elements in *CheckResult* (obviously, *CheckResult* is the set of the observations on the output models), and *expct* is the expected result of the observations on the transformation of *IM* (i.e., what forecasted by the *oracle*). The test is passed if $check(\mathsf{mtt}(IM)) = expct$. A *test suite* is a set of test cases.

We classify the MTT tests on the basis of the *intent* with which they are carried out (taking also inspiration from [43]).

Conformance tests are made with the intent of verifying if the MTT results belong to the target. In general this means checking that the textual artifacts produced by the transformation have the required structure (e.g., a folder containing at least two files and no subfolders) and that the various files have the correct form (e.g., they are correct with respect to a given BNF or XML schema).

Referring to Fig. 5.1, conformance tests are done with the intent of verifying that all the elements belonging to the *Source* are transformed by *mtt* into elements of the *Target*. The *check* function must verify some properties characterizing the target. For instance, in the case of a transformation producing Java programs, a check may verify that they are syntactically correct/they use only standard API/all their identifiers are lower case; whereas for an MTT producing HTML documents a test may check that they are correctly visualized by a specific browser[2] or that they describe a web site respecting some accessibility requirements.

[2]obviously this test will be performed by a human being

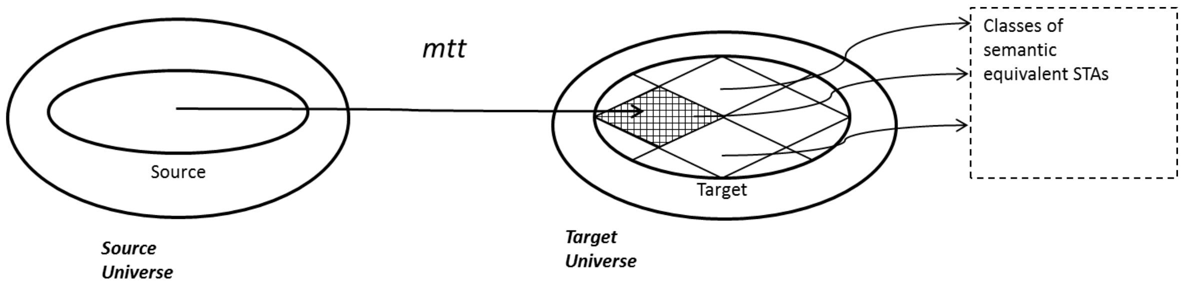


Figure 5.10: Transformation function – Semantic test

If a conformance test fails, then we firstly have to look at the constraints of the source and target definitions to see if they are too weak/too strong, and later we have to examine the *mtt* design and implementation. To better clarify this point consider the following example:

> ***U2SQL: Conformance test*** in this case, if the constraints over the transformation source do not prohibit class names with whitespaces, then a substantially correct transformation will transform these element names into wrong table identifiers, and conformance tests will fail. The correct developer action is to refine the definition of the source so to forbid UML element names containing whitespaces.

In a conformance test case $(IM, check, expct)$, *check* and *expct* do not depend on *IM*, because these two test case ingredients are built only by looking to the definition of *Target* and the transformation requirements.

Referring to the U2OWL case, an error discovered by conformance tests is the following: the transformation forgot to add the closing tag to the definition of the OWL class corresponding to a UML class. Trying to open with Protege[3] the OWL file produced by the transformation of whatever trivial model (i.e., containing at least a UML class) we are informed of an exception occurred during the parsing of the input file (see the complete test suite in Appendix A).

Semantic tests are made with the *intent* of verifying that the target of the MTT has the expected semantics. Referring to the Fig. 5.10, semantic tests are done with the intent of verifying that elements belonging to the *Source* are transformed into elements of the *Target* belonging to the right class of equivalent STAs (class of equivalent STAs are depicted with diamonds in the picture). The *check* function using to build the test cases has to verify semantic properties typical of the target nature, using methods and techniques again typically of that context (e.g., SQL DDL scripts, Java programs, OWL ontologies)

To clarify the concept we give some examples: (1) if the transformation target are SQL DDL statements, then a semantic test may check if the SQL statements are valid in the light of the

[3]http://protege.stanford.edu/

objects in the database, for example checking if the tables and columns referenced in a SQL statements actually exist in the database (e.g. in a foreign key definition); (2) if the transformation target are Java programs, then a semantic test may verify if some class operation has the required behaviour by means of classical tests or that an interactive program may execute some specific scenario; (3) in the U2OWL case the meaning of a OWL file is an ontology made of semantic categories (OWL classes), containing individuals, and of inclusion/sub-typing relationships between categories, thus the semantic tests may check if the represented ontology has all the OWL classes described in the UML input model, or that an OWL class has all the required individuals, or that an OWL class is a sub-class of another one.

Semantic tests should be built taking into consideration all the techniques and methods already existent for the nature of the *Target*. For instance, if the *Target* is the set of all the Java projects for desktop applications, then we can employ the techniques used to test (the semantic of) a Java program.

In the case of U2SQL we can exploit the possible RDBMS capability of semantic checking, or we can check the existence of database elements querying appropriate system tables.

Referring to the U2OWL case, an error discovered by semantic tests is the following: the input model is composed by a class diagram containing a hierarchy of classes each one stereotyped with ≪category≫ and an object diagram containing only one object instance of one of the leaf classes. The output produced must contain an ontology in which are described the same class hierarchy and an individual belonging to a leaf class. A semantic test could be the "realization of an individual"[4] reasoning task that may be performed by a reasoner like FaCT++ [112].

Semantic tests depend on the source model. Indeed, the form of the check function depends on what elements are contained in the source model. In some cases the expected results can be automatically derived from the source model Semantic tests are very important because allow to find the most serious failures, for instance when the transformation requirements are not met, even if the transformation result is a running Java program or an OWL ontology correctly shown by Protege.

Textual tests are made with the *intent* of verifying that the textual elements comprising the STA target of the MTT have the required form. Referring to the Fig. 5.11, textual tests are done with the intent of verifying that each element belonging to the *Source* is transformed into the right element of the *Target*. The *check* function must verify that the result of the transformation considered as a pure textual artifact has the right form. Textual tests depend on the input models, similarly to the semantic tests.

For example: (1) if the *Target* is JAVA code, then a textual test may verify if there is a correct number of files, in the right position in the file system and with the right names; (2) if the *Target* is an ontology written using OWL, then a textual test may verify if it contains the right number

[4]that is, find all classes which the individual belongs to, especially the most specific one

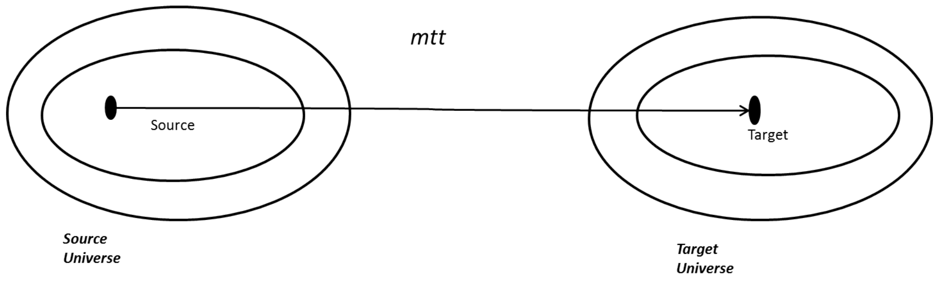


Figure 5.11: Transformation function – Textual test

of class structure definitions; (3) in the U2SQL case an error discovered by textual tests is the following: the target contain the string "AT char(60)" in correspondence of an attribute *AT* typed by the UML basic *string* type in the source model, instead of the string "AT char(64)" (as required by the design). In this case the string found in the target is correct with respect to a given BNF, but it is not the expected one (in our example the string "AT char(64)"). We want to emphasize that this kind of errors are not revealed by the conformance test.

In the following we shown some textual tests that can be performed in the case of the U2SQL transformation:

U2SQL: Textual test

[**Table name**] for each construct "CREATE TABLE *X*" in the *Target*, there must exist in the *Source* a class stereotyped by ≪table≫ and named named *X*.

[**Column Name**] for each construct "*attr data-type*" inside a "CREATE TABLE *X*" construct in the *Target*, there must exist in the *Source* a class stereotyped ≪table≫ and named *X* with an attribute *attr*.

[**Column Name1**] for each construct "*attr data-type* NOT NULL" inside a "CREATE TABLE *X*" construct in the *Target*, there must exist in the *Source* a class stereotyped ≪table≫ and named *X* with an attribute *attr* stereotyped by ≪key≫.

...

5.4.2 Unit Testing

A non-trivial MTT will be built by composing many sub-transformations, that transform different - but potentially overlapping - parts of the input models; The various sub-transformations may be arranged in a decomposition diagram as it explained in Sect. 5.3.3.

Thus, we can decompose the whole transformation in parts, each one composed by the sub-transformations belonging to one of the sub-trees, and test these parts separately. For example

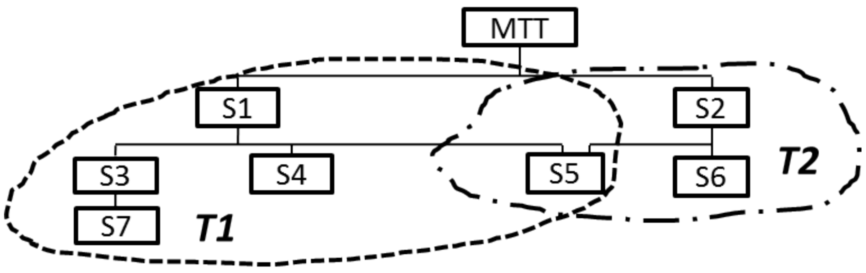


Figure 5.12: A sample of MTT structure

in a transformation whose decomposition is showed in Fig. 5.12, we can test separately the parts of the transformations corresponding to the sub-trees *T1* and *T2*. The testing of the transformation parts will use test cases built with model fragments, but it should be checked that still they satisfy the restrictions imposed by the definition of *Source*, i.e., they may be extended to become models in *Source*. The developer should also choose a decomposition that allows to have transformation parts that generate self-contained structured textual artifacts[5], to allow the semantic testing. Using this approach we can build a kind of unit testing of a MTT. Indeed, each part of the transformation can be tested separately.

U2SQL*: *Unit Test

In the case of the U2OWL transformation there is an obvious partition of the whole transformation in two sub-transformations as it is shown in Fig. 5.13.

These sub-transformations are the sub-transformation **T2** that regards only tables without foreign-keys, and the sub-transformation **T1** that regards the whole transformation.

Other partitions require the building of functional stubs, like for example, the sub-transformation composed by the nodes under the node labelled **TAttributes** (in this case we unit test only the part of the whole transformation regarding the class attributes without the ≪key≫ stereotype. Thus, building proper functional stubs we can unit test almost all the MTT.

[5]otherwise build proper functional stubs

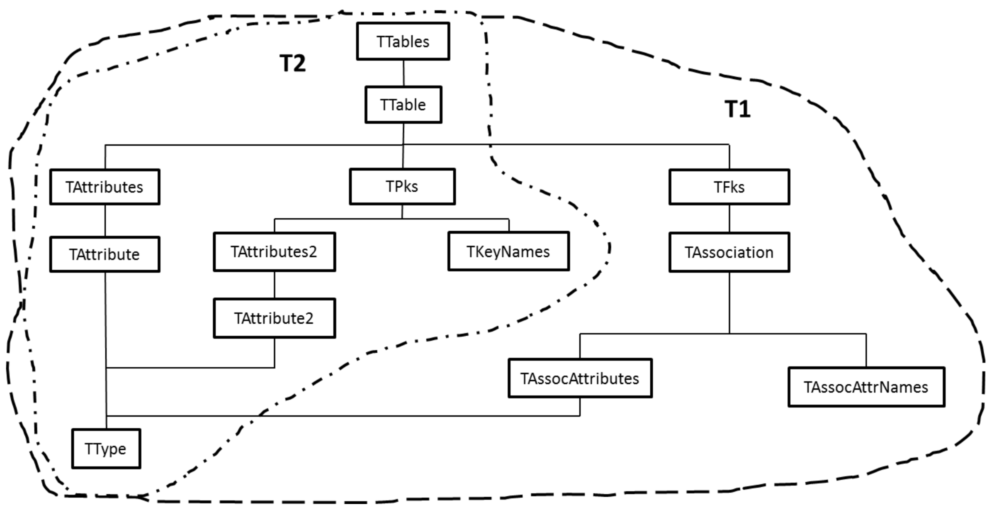


Figure 5.13: U2SQL: unit test

5.4.3 Transformation Test Adequacy Criteria

A *test adequacy criterion* is a criterion that drives the selection of *characteristics* for input models that will be used to generate the test models needed to build the test suites (adapted from the definition found in [51]). We use model characteristics instead of whole models for criteria definition, because defining whole models requires a substantial effort and to fix a large number of details not relevant (e.g., to require to have a UML model with a class with a given stereotype and 3 operations and no attributes versus to have to produce a complete model defining also the names, the parameters, and the result types of such operations).

The *coverage* is the ratio between the number of characteristics identified by a test adequacy criterion that can be found in the input models of a test suite and the total number of characteristics identified by the same test adequacy criterion.

In the following we describe three adequacy criteria. Let *S* be the set of the stereotypes and tagged values comprising the UML profile used for defining *Source*; let *CD* be the set of the UML constructs/diagrams allowed to appear in the models in *Source* considered relevant by the transformation developer; and let $M = S \cup CD$.

Referring our running example U2SQL, S={≪table≫, ≪key≫ } and CD={class, association}. In this case the developer did not considered relevant to insert in CD also the attributes.

Criterion 1. The predicates on *Source* defining this criterion are: $pred_x$: *Source* $\rightarrow Bool$ defined by $pred_x(Mod) \Leftrightarrow x$ appears in Mod, for $x \in M$, $Mod \in$ *Source*; thus we have the same number of predicates and elements of *M*. Each test model must contain at least one of the elements of

M. Starting from this criterion we can define other criteria, simply requiring that each predicate considers more than one element of *M*, e.g., if $x, y \in$ *M* and $x \neq y$, then we can define a predicate $pred_{x,y}$: *Source* $\rightarrow Bool$ such that $pred_{x,y}(Mod) \Leftrightarrow x, y$ both appear in Mod. So, for example in our case the set *M* is composed by four elements thus, we need, at least, to define 4 $\times$ 3 predicates.

U2SQL: Definition of test models following the adequacy criterion 1 In this case we have four predicates, but, given the well-formedness rules Sect. 4.2 which requires that each class stereotyped ≪table≫ must have at least one attribute stereotyped ≪key≫, is not possible to have one model for each predicate (so to have models in which only one predicate is true).

Moreover, we build other test models in which these four predicates will be true more than one time (say three, for example). Each of these models must contain at least three elements of M, in all the possible configurations.

Table 5.3 shows all the models built following this criterion.

Criterion 2. This adequacy criterion takes into account the features of the elements in *M*, and is defined by the set of predicates that evaluate the presence in the input models of a set of relevant combinations of such features. The developer must consider that some combination of them may be not allowed by the definition of *Source*. Each test model must contain at least one of these sets of relevant combination of features. In the U2SQL case, the features of class (that belongs to M) allowed in the *Target* (just the attribute) will contribute to the criteria, for instance, with predicates checking the presence of a class with one attribute and of a class with attributes of all the primitive types.

U2SQL: Definition of test models following the adequacy criterion 2

In this case the only relevant combination of features of *M* are the class attributes of each primitive types. Given the definition of *Source* we take into consideration the type variations of attributes stereotyped by ≪key≫ plus type variation of attributes without the stereotype ≪key≫.

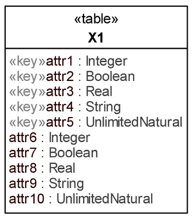


Figure 5.14: U2SQL Criterion 2 test models

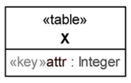
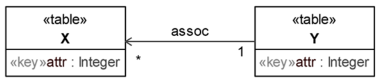
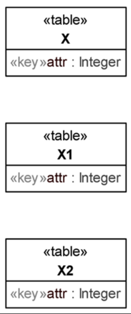
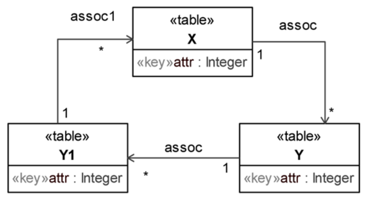
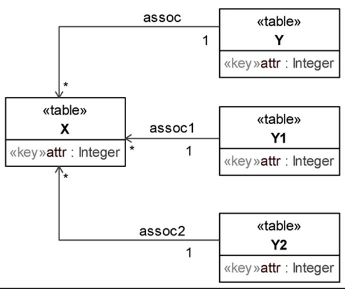
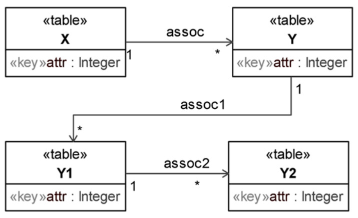
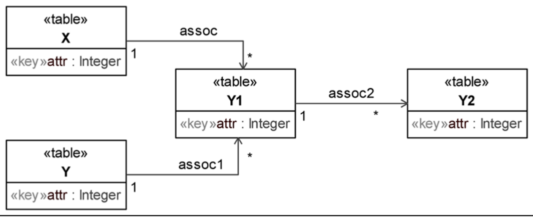
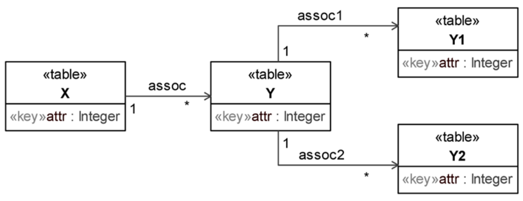
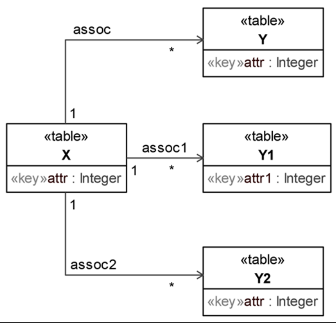
	 «table» X «key»attr : Integer assoc * 1 «table» Y «key»attr : Integer
 «table» X «key»attr : Integer «table» X1 «key»attr : Integer «table» X2 «key»attr : Integer	 assoc1 * «table» X «key»attr : Integer assoc 1 * 1 «table» Y1 «key»attr : Integer assoc * 1 «table» Y «key»attr : Integer
 assoc 1 «table» Y «key»attr : Integer * «table» X «key»attr : Integer assoc1 * 1 «table» Y1 «key»attr : Integer * assoc2 1 «table» Y2 «key»attr : Integer	 «table» X «key»attr : Integer assoc 1 * «table» Y «key»attr : Integer 1 assoc1 * «table» Y1 «key»attr : Integer assoc2 1 * «table» Y2 «key»attr : Integer
 «table» X «key»attr : Integer assoc 1 * «table» Y1 «key»attr : Integer assoc2 1 * «table» Y2 «key»attr : Integer «table» Y «key»attr : Integer assoc1 1 *	 assoc1 1 * «table» Y1 «key»attr : Integer «table» X «key»attr : Integer assoc 1 * «table» Y «key»attr : Integer 1 assoc2 * «table» Y2 «key»attr : Integer
 assoc * «table» Y «key»attr : Integer 1 «table» X «key»attr : Integer assoc1 1 * «table» Y1 «key»attr1 : Integer 1 assoc2 * «table» Y2 «key»attr : Integer	

Table 5.3: U2SQL Criterion 1 test models

Criterion 3. The adequacy criterion takes advantage on the way we give the MTT design. The design of each transformation function is given by means of relevant source-target pairs. Each pair is composed by a left side, that shows a template, and a right side that shows the result of the application of this function on model fragments obtained instantiating such template. So, criterion 3 is defined by the set of predicates checking that the various templates are instantiated on the input models. Each test model must contain at least one of the templates showed in the left side of the source-target pairs.

U2SQL: Definition of test models following the adequacy criterion 3

To build models following this criterion we can only use the MTT function definitions in which was not given the function definition.

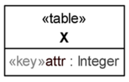
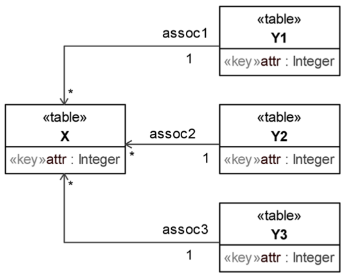
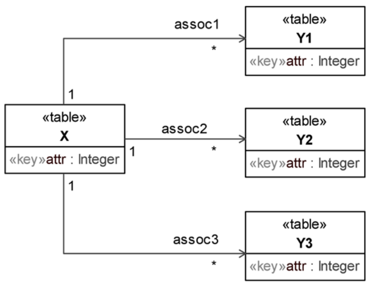
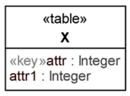
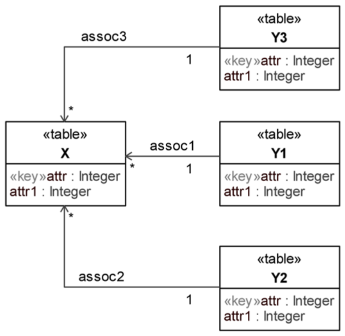
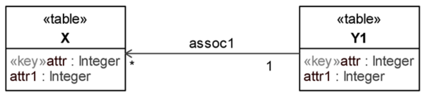
	 «table» X «key»attr : Integer «table» Y1 «key»attr : Integer «table» Y2 «key»attr : Integer «table» Y3 «key»attr : Integer assoc1 assoc2 assoc3 1 *
 «table» X «key»attr : Integer «table» Y1 «key»attr : Integer «table» Y2 «key»attr : Integer «table» Y3 «key»attr : Integer assoc1 assoc2 assoc3 1 *	
 «table» X «key»attr : Integer attr1 : Integer «table» Y1 «key»attr : Integer attr1 : Integer «table» Y2 «key»attr : Integer attr1 : Integer «table» Y3 «key»attr : Integer attr1 : Integer assoc1 assoc2 assoc3 1 *	 «table» X «key»attr : Integer attr1 : Integer «table» Y1 «key»attr : Integer attr1 : Integer assoc1 1 *

Table 5.4: U2SQL Criterion 3 test models

5.4.4 Transformation Testing Automation

We consider two levels of testing automation: (1) automate the generation of the artifacts needed for testing and (2) automate the execution of testing. In the former case, the check function and sometimes also the expected results are automatically generated from the elements contained in the source model. In the latter case the execution of testing, and sometimes also the retrieval of testing results, can be partially, or totally automatized.

Automate test generation. If the generation of testing artifacts ***can*** be described in an algorithmic way, the needed artifacts can be produced during the execution of the MTT, to be more precise, by an extension of MTT that together with the target produces also the test cases components, without adding any information to the source models. This is possible when the structure of the artifacts and the input data for the test cases depend only on the content of the source models. For example, if we consider the U2Java case, we have that the source models may contain persistent classes (those stereotyped by ≪store≫), then the generated persistent entities may be tested generating a JUnit test that creates number of entity instances and then retrieve them.

If the generation of the needed testing artifacts ***cannot*** be described in an algorithmic way, we can try to add some parts in the source model and extend the transformation to map them into tests. For example in the case of U2Java, we do that by inserting in the source model test classes and test operations, that drive the generation of executable JUnit tests in the target code (see Sect. 6.2.3)

Automate test execution. Automating the execution of testing becomes an important issue when the number of test cases grows, and should be supported by tools. Depending on the nature of the transformation target, tools designed expressly for automate the execution of testing may already exist (e.g., JUnit for Java, see Sect. 6.2.3) . After that the testing artifacts are generated (automatically or by the transformation developer), a specifically designed application can execute the test suites and, in some cases, can generate reports containing the result of testing. For example, if we consider U2Java, the testing execution can be automated by means of Maven [2]. All the test generated artifacts consist, in this case, of Java code and configuration files placed both in the same project in which the application code is generated, but in a separate folder, that Maven recognizes as the tests container. The developer only needs to activate the build process. During the build process tests are executed and report files are produced.

5.4.5 Transformation Testing Method

Our testing method prescribes that the test suites must be built manually, although following our transformation test adequacy criteria (see Sect. 5.4.3) and using test of the kinds described in Sect. 5.4.1.

We write the test suites manually, because:

Conformance Test	**Textual Test**	**Semantic Test**
may be problematic	**the most suitable**	may be problematic

Table 5.5: Unit Test: tests kind appropriateness

Criterion 1	**Criterion 2**	**Criterion 3**
restricted to the transformed model fragments	restricted to the transformed model fragments	**the most suitable**

Table 5.6: Unit Test: build input models

- writing test suites manually help us in refining/validating the model transformation design, especially defining semantic tests.
- automate the generation of test suites using our method to design the model transformation is quite difficult, if not impossible, given that it is not expressed in the terms of the source and target meta-models;
- building test input models manually, makes easier for the developer to understand the test results and find errors in the model transformation.

Our testing method requires to first execute the unit tests (*Unit test*), then execute the the test over the whole transformation (*Whole transformation test*).

Unit Test in this case the subject of testing may be a sub-transformation which may produce ill-defined fragments or fragments without explicit semantics. Thus, the most suitable kind of testing is the *textual test* whereas *conformance test* and *semantic test* may be problematic. Indeed, *conformance test* fail with ill-defined fragments and *semantic test* fail with fragments without explicit semantics. *Semantic test* and *conformance test* may still be used building proper functional stubs.

In this case, the most suitable test adequacy criterion is the *criterion 3*, given that using this criterion we build input models containing only the model elements belonging to the domain of the transformation functions composing the sub-transformation object of the test. *criterion 1* and *criterion 2* may still used but they must be restricted to the transformed model.

Table 5.5 and Table 5.6 summarize how to choose the test kind and the test adequacy criteria in the case of unit test.

Whole transformation test in this case, the most suitable kind of testing is the *conformance test* because it reveals the largest number of errors, given that tests of this kind are able to detect all MTT implementation errors related to generate text not following the required concrete syntax, e.g., forgetting to close a parenthesis or forgetting a space between two keywords (which in our experience are very common). Tests of *Semantic test* kind are able to detect more serious errors

Conformance Test	**Textual Test**	**Semantic Test**
reveal the largest number of errors	the less suitable	reveal most serious errors

Table 5.7: Whole Test: tests kind appropriateness

Criterion 1	**Criterion 2**	**Criterion 3**
suitable	**suitable**	the less suitable

Table 5.8: Whole Test: build input models

due to the requirements understanding or design/implementation errors. Finally, *textual test* is the less suitable.

The preferred test adequacy criteria are *criterion 1* and *criterion 2*, whereas *criterion 3* is obviously the less suitable.

The table Table 5.7 and Table 5.8 summarize how to choose the test kind and the test adequacy criteria in the case of whole transformation test.

5.4.6 Regression Test

A very simple oracle functions is the one that compares the output of a specific run of the model transformation with the expected output, and in case of textual targets the comparison is made without considering white spaces and line breaks. The expected output is generated by the transformation itself and its correctness is assessed by the transformation developer (possibly with the help of available tools, depending on the nature of the produced artifacts); she/he corrects the transformation until reaching a satisfactory result (i.e., a correct output).

This process is feasible because the expected output is human readable.

Moreover, to perform the regression test the transformation must produce the same artifacts (not only semantically equivalent, but syntactically the same) for the same test model. Therefore a particular attention must be paid to the transformation implementation, especially when model elements does not have an intrinsics order (i.e. the class attributes, or class operations in the U2Java transformation).

Regression testing is useful in the case in which new features are added to model transformation or it is refactorized; in the other cases new versions of the expected projects are required.

5.4.7 Testing of Input Models

Semantic tests (described in Sect. 5.4.1) and the techniques used to automate the test generation when the generation of testing artifacts cannot be described in algorithmic way (described in Sect. 5.4.4) may be used also also as a way to perform some tests on the input models. Using these techniques we can insert in the input model test classes containing test operations whose behaviour is known and operation post-conditions to verify that the behaviour is the expected one. If the behaviour of the operation is very simple, the probability of erroneous definition of the model is lower than the probability of erroneous definition of the transformation (qualitatively speaking), thus we can consider the transformation incorrect also if in this case we cannot totally exclude errors in the model. In case of success, we gain in confidence on the correctness of the transformation, except in the case of two errors, one in the source model and the other in the transformation, that compensate each other, but this can be considered a really rare case, also because the errors in the two fields are of a very different nature.

Once that the confidence on the correctness of the transformation has reached an acceptable level, we can consider that failure of such test should denote a problem in the model, since we are now assuming that the transformation is correct. Moreover, after that is assumed that the transformation is correct, we can insert in the model test operation whose behaviour is not very simple. A fail of this kind of operations reveal an error in the input model since the model transformation is correct. This can be done not only when the target of the transformation is executable, but also in the other cases. For example in the case of the U2OWL transformation, test classes and test operation may be transformed into Java programs that, using Java libraries as OWL API [6], may execute semantic tests over the ontology generated by the transformation.

This goes in the direction of using only models also for what concern testing. A user of the transformation may completely forget about the structure and the technology used in the target of the transformation, and concentrate herself/himself on producing and checking the input models, thus increasing the level of abstraction used in the development of software applications, as claimed by MDD.

5.5 Transformation Implementation

From a practical point of view, the implementation phase of the *MeDMoT* involves the following actions:

1. code the diagnostic metamodel;

[6]http://owlapi.sourceforge.net/

2. implement the check of the well-formedness of the source model and build the diagnostic model, (this is done using a model to model transformation, see Sect. 5.3.1);
3. implement the re-factorization of the input model (this is done using a model to model transformation language, see Sect. 5.3.2);
4. implement the TGT Sect. 5.3.3 (this is done using a model to text transformation language).

The *MeDMoT* give guidelines and patterns to follow in carrying out these actions. Moreover, it prescribes which model transformation languages and which tools to use.

Before describing these guidelines is necessary to list the chosen model transformation languages and supporting tools. The process that has lead us to select such items and its results are described in the following section.

5.5.1 Model Transformation Languages and Tools Selection

For applying *MeDMoT* we need:

1. a tool to code the diagnostic meta-model;
2. a model to model transformation language and a supporting tool to perform the well-formedness check;
3. a model to model transformation language and a supporting tool to perform the refactoring;
4. a model to text transformation language and supporting tool to perform the TGT.

Summarizing we have to select: (1) a tool to write the diagnostic model, (2) a model to model transformation language and its supporting tool (these are needed for well-formedness check and the refactorization of the input model), (3) a model to text transformation language and its supporting tools.

To make these choices we have defined a set of criteria, distinguishing between those for the selection of model transformation languages and those for selection of supporting tools. Finally, the choices were made based on the results of some experiments on the selected languages and tools.

Model Transformation Language Selection

Selecting the right model transformation languages is a problem itself, indeed there are a lot of model transformation approaches and languages [104]. We have selected the model transformation languages on the basis of their features [61], their diffusion, their standardization and the existence of supporting tools.

Tools Selection

We have selected the tools taking into account the external and internal quality of tools as they are defined in ISO/IEC 9126 [40]. Moreover, we took into consideration the cost of supporting tools. Indeed, we prefer open source solution to avoid lock-in problems and maintain low their adoption cost. Another characteristic of the tools taken into consideration during the selection process, is the integrability of tools, preferring those that are already integrated (i.e., belonging at the same tools framework) or that can be easily integrated. Obviously, the interoperability among selected tools is an essential feature. Finally, tools that are implementation of standards are preferred over the others.

Our attention has been focused on one of the most widespread open source frameworks: the Eclipse Platform [12]. In this context there is the *Eclipse Modeling Project* [11] (EMP) that contains all the official projects of the Eclipse foundation dealing with model based developing technologies. The EMP contains, among the others, the following projects (only projects of interest are listed here):

- **Abstract Syntax Development**
  - *Eclipse Modeling Framework* [10], (EMF) it is a modelling framework that provides tools and runtime support that enable viewing and editing of the model. Moreover, it provides the ecore metamodel (a metamodel similar to MOF [20]) for describing models and a framework to serialize and de-serialize models in XMI [21] format.
- **Model Development Tools**
  - *OCL* [18], it is an implementation of Object Constraint Language (OCL) OMG [24] standard for EMF-based models
  - *Unified Modeling Language 2.x* [37], (UML2) it is an EMF based implementation of the OMG UML 2.x [25] metamodel.
  - *Papyrous* [26],it is a graphical editor for UML models. It's based on EMF (at the time in which it was carried out tools selection, it was in incubation[7] phase).

[7]incubation refers to a phase of the Eclipse Development Process. Purpose of this phase is to create a fully functional project.

- **Model Transformation**
  - *Query/View/Transform Operational* [28] (QVT) it is a partial implementation of the Operational Mapping Language defined by OMG MOF Query/View/Transformation [23].
  - *Query/View/Transform Declarative* [27] (QVTd) it is a partial implementation of the Core and Relational Language defined by OMG MOF Query/View/Transformation [23]. It is in incubation phase.
  - *Acceleo* [1], it is an implementation of OMG MOF Model to Text Language (MTL) [22] plus an IDE and a set of tooling to simplify the production of model to text transformations.
  - *Java Emitter Template* [15], it provides a code generation framework based on a model to text transformation language based on templates.
  - *Xpand* [39], it is a language specialized in code generation for EMF models.

Among these tools we can find all the tools we need. They are open source and already integrated, indeed, they are based on the Eclipse platform. Moreover, they are interoperable, because they are all EMF based.

To write the input model, the selected tool was the Papyrous editor, but after some experiments and considering that the project is (and was, at the time the selection was made) in incubation phase, we have decided to experiment a commercial tool for writing UML models. After a short exploration and based on past personal experiences, we decided to select the MagicDraw[8], given that it supports the UML2 metamodel. Moreover, this tool is easily integrable in Eclipse. This can be considered a temporary solution until the Papirus UML editor will become more mature and stable. A particular attention has been devoted to verifying the support of UML profiles by the tool.

To do the well-formedness check and the refactoring as explained in Sect. 5.3.1 and in Sect. 5.3.2 we need a model to model transformation tool and a tool to write the diagnostic metamodel. In the EMP there was two model to model transformation tools not in incubation phase: QVTo and ATL. We chose the ATL model to model transformation language for the following reasons:

- ATL was a tool more mature than QVTo (the QVTo tool had reached the version 1.0 only in November 2008 and the version 1.0 of the OMG specification of the MOF Query/View/ Transformation dates back to April 2008);
- in literature we found more reference on ATL [75, 76, 77, 105, 56];
- we found more examples of use of the ATL tool and transformation language than example of use of the QVTo tool and language;

[8]`http://www.nomagic.com/products/magicdraw.html`

- ATL is itself a QVT-like model transformation language [77, 78].

To write the diagnostic metamodel we chose to use EMF, given that it is often a simple model.

To do the model to text transformation, among Acceleo, Java Emitter Template and XPand, we chose Acceleo basically because it is an implementation of an OMG standard model to text transformation language. Moreover, the Eclipse Acceleo tool, offer an IDE [9] with context assist, syntax highlighting and all the features of a modern IDE.

5.5.2 Well-Formedness Check implementation

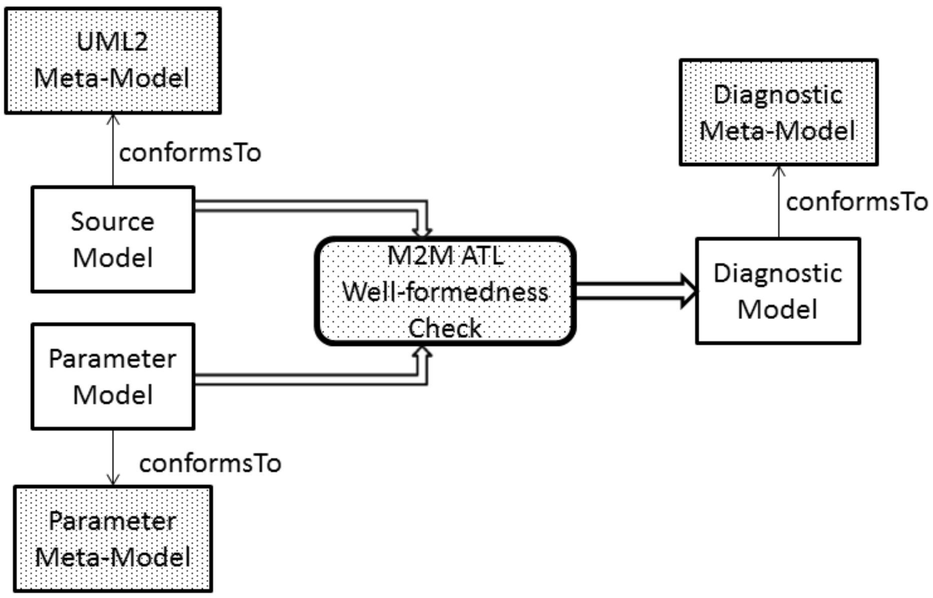


Figure 5.15: Well-formedness check implementation architecture

In this section we will explain how to implement the well-formedness check using the Eclipse environment and a model to model transformation written using ATL in a way similar to that suggested in [53].

Fig. 5.15 shows the well-formedness check implementation architecture. The well-formedness model to model transformation is implemented in ATL using the Eclipse ATL tooling set.

This transformation has two inputs and one output. The inputs of the transformation are:

1. the input model, conform to the UML2 metamodel and

[9]Integrated Development Environment

2. the well-formedness transformation parameter model, a model that contain parameters that can filter which well-formedness rules must be evaluated by the ATL transformation. It is instance of the parameter metamodel.

This parametrization of the transformation may be useful when the total number of constraints that must be evaluated during the transformation is very high, thus having a high computation cost. The activation of the constraint evaluation can be parametrized, so as to choose what constraints evaluate each time that the diagnostic model is generated. Output of the transformation is the diagnostic model instance of the the diagnostic metamodel (see Sect. 5.3.1).

Before implementing the ATL transformation we have to write the diagnostic metamodel, that is an EMF model, to do this task we can use the Eclipse Ecore Tools [9]. To write the diagnostic metamodel we must create an empty EMF project and then create an Ecore diagram. In this diagram can create the Error abstract class and all the derived classes as explained in Sect. 5.3.1.

After having written the diagnostic metamodel, we must write the parameter metamodel. Since it is also an EMF model, we can write it in the same way we write the diagnostic metamodel.

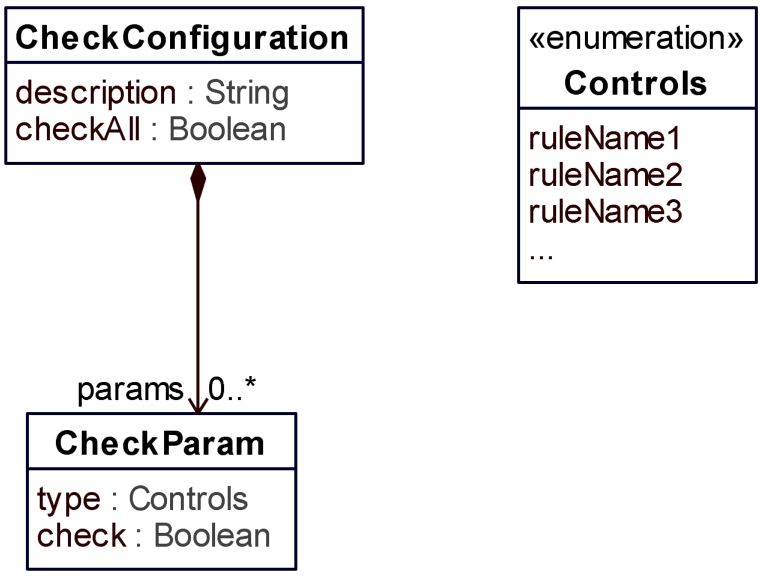


Figure 5.16: Example of a Well-formedness Parameter Metamodel

Fig. 5.16 shows an example of a parameter metamodel.

The *CheckConfiguration* element may contains zero or more *CheckParam* elements. Each instance of the *CheckParam* element has a flag, *check*, which indicates whether the corresponding well-formedness check rules must be performed or not. The *Controls* enumeration contains literals corresponding to all the possible checks, and it is used for typing the *type* attribute of the

CheckParam class. May be helpful to name the classes of the diagnostic metamodel derived from the *Error* class, with the literals contained in this enumeration. Moreover, the *CheckConfiguration* element has the *checkAll* flag to indicate that all the rules must be evaluated.

Having the parameter metamodel we can write one or more parameter models using Eclipse. The easiest way to write a parameter model is to open the parameter metamodel with the *Sample Ecore Model Editor*, then select the metamodel root element (in out example the *CheckConfiguration* element) and using the context sensitive menu, select the *Create Dynamic Instance...* and give the name of the file that will contain the parameter model (e.g., CheckAllConfiguration.xmi).

Finally we can write the ATL transformation that implement all the well-formedness rules. In the following we give also some rule design pattern to facilitate the well-formedness rule implementation with ATL.

Before giving guidelines on how to write the ATL transformation that implement the well-formedness rules, we summarize the basics of the ATL model to model transformation language:

Writing the Well-formedness Check ATL Transformation

Starting from the list of well-formedness rules listed in the design of the well-formedness check, we have to implement a model to model transformation using ATL.

For each well-formedness rule, the well-formedness model transformation must evaluate a logical constraint over the input model and, if the corresponding configuration parameter allow it, must generate a specific element in the target model (i.e., an element of the diagnostic model). We chose to create target elements using the ATL construct "called rules" [4]. An ATL called rule must be invoked from and ATL imperative block, that we decided to specify into a matched rule [7]. Moreover, we used ATL helpers [5] contained in ATL libraries [6].

The header section of the main module of the transformation must be something like this:

```
module WellFormednessCheck;
create OUT: MM1 from IN: UML2, IN1: MM, PROFILE: UML2;
```

In this example we have reference to the following metamodels: MM1 that is the diagnostic metamodel, UML2 that is the UML2 metamodel and MM that is the parameter metamodel.

All the matched rules have the following form:

```
rule matched_ rule_ name {
   from
     s: UML2!Model_ element,
     p: MM!CheckConfiguration
   do {
        if(s.name_of_the_design/stereotypes_helper and p.name_of_configuration_helper ))){
          thisModule.name_of_called_rule_that_generate_the_corresponding_target_element;
        }
        ...
}
```

Where:

- *Model_element* is the source model element that must be matched;
- *name_of_the_design/stereotypes_helper* is the name of a contextualized helper (helpers may be contextualized to a model element, see [5]) that evaluate to true if the well-formedness constraint evaluate to true;
- *name_of_configuration_helper* is the name of a contextualized helper that evaluate to true if this rule is enabled in the parameter model;
- *name_of_called_rule_that_generate_the_corresponding_target_element* is the name of a called rule that can create an element of the right type (types of elements of the target model, are defined in the diagnostic metamodel) in the target model.

5.5.3 Refactoring

As explained in Sect. 5.3.2 the refactoring process of the input model is composed by one or more model to model transformations that must be executed in a well defined order. Each transformation is implemented using the ATL transformation language taking advantage of the *refining* mode of the ATL compiler [8, 105].

Using this execution mode of ATL only the input model elements involved by the transformation rules are changed, the others remains unchanged. Moreover, using the *refining mode* the source and the target model may be the same.

The ATL refining mode implement the in-place strategy [10] starting from the ATL2006 ATL compiler version, and in the ATL2010 compiler version is supported also the deletion feature using an explicit drop *keyword*. Using the refining mode, some of the advanced ATL compiler features remain unsupported.

Since the refactoring operations may be of very different nature, we can not give implementation pattern as in we do in Sect. 5.5.2.

[10]With this strategy the target model is not constructed incrementally starting from an empty one, rather the starting target model is the whole source model, then transformation rules are executed and the model elements that are not manipulated (created, updated or deleted) remain unchanged [105].

The transformation rules must by written following the transformation cases as explained in Sect. 5.3.2.

5.5.4 Model To Text Transformation

To implement the model to test transformation using Acceleo, following the design of the transformation we have to create as many templates as are the transformation functions. The templates have as parameters of the elements of the UML2 metamodel. Thus, the developer must extrapolate what elements of the UML2 metamodel are to be used as parameters instead of those used in the definition of the model transformation function. In the model transformation design, these parameters can be of two kinds depending on the type of design that has be done: *User Level Design* or *Developer Level Design*. In both cases it is required a deep knowledge of the UML2 metamodel, but in the latter case this is obviously more simple.

Moreover, from each transformation case the developer must infer which elements of the metamodel must be selected and if necessary used as parameters for other transformation function (usually shown in the left side of the transformation case). Each element selection can be implemented using Acceleo queries.

Since the model to text transformation design is quite independent from the kind of model transformation language chosen for the implementation we do not have direct indication (from the design) on how to architect the Acceleo project, leveraging the Acceleo features.

Templates can be grouped into modules, according to the metrics normally applied to JAVA classes. Thus, modules should not contains a huge number of templates. Modules should be designed for re-use.

Input models can contain invariant pre and post-conditions (see Sect. 4.4) expressed in the OCL language and contained in the models as text. If we need to transform them (for example in JAVA language) using Acceleo templates, first we have to transform them in instances of a metamodel that can be used by Acceleo templates (that are substantially all the EMF based metamodels). This transformation can be done using Acceleo JAVA service that, using the OCL support of the Eclipse platform [18], can parse the OCL expressions and transform them into a instances of the OCL metamodel that in turn can be used by other Acceleo templates to complete the process of transforming text representing OCL expressions (like invariant and/or pre-postconditions) into JAVA code (see Sect. 6.2.4, for an example of use).

Moreover, Acceleo JAVA services are useful to transform the text representing method bodies into JAVA.

5.5.5 Testing

Despite the chosen testing techniques are quite general (see Sect. 5.4), the implementation of these techniques, is dependent on the form of the target of the model transformation, i.e., is dependent on the type of structured textual artefacts that are produced by the model transformation. See Sect. 6.2.3 for a description of the implementation of these testing techniques when the target is a Maven project of a JAVA desktop application.

5.6 Conclusion

In this chapter we have outlined *MeDMoT*, our method to develop model to text transformations. This is a method to engineer model transformation development life-cycle, indeed *MeDMoT* cover all the phases needed to develop a model to text transformation (using well-founded software engineering principles), from requirement definition to model transformation implementation. First, we have described a method to capture MTT requirement.Then, we have presented a method to design a MTT based on the concrete syntax of source and target of MTT and following a specific architecture. Moreover, we have paid a particular care in defining a method to test MTT, giving: (1) a classification of MTT testing based on the intent with which a developer should test the MTT, (2) three new model transformation test adequacy criteria (also them based on the concrete syntax and semantic of the source and the target of the transformation) and (3) a way to calculate test coverage.

In addition, in this chapter we have given guidelines on how to implement the model transformation, selecting the model transformation languages needed, for the model to model transformations and the model to text transformation composing our MTT, and the tools eco-system to be used during the MTT implementation.

Summarizing, MeDMoT has the following features: (a) deals in integrated way with all the MTT developing phases; (b) provides guidelines on how to define the requirements; (c) guides the design of the MTT; (d) helps in modularize the design of the MTT; (e) provides a lightweight notation to be used in the MTT design; (f) gives guideline on how to implement the design MTT (suggesting also the tools and the IDE that are to be used).

Chapter 6

Case Studies

To prove in practice the *MeDMoT*, we have selected two case studies, a small/one and a quite large one (another toy-level application U2SQL has been used for illustrating *MeDMoT* in Chapter 5):

U2OWL: transformation of UML models of ontologies into the corresponding definitions written using the RDF/XML concrete syntax of the OWL language;

U2Java: transformation of UML models of the design of JAVA desktop applications into JAVA desktop applications built using the Spring framework as glue-framework, and JPA with Hibernate[1] as persistence provider, in the Data Access Layer.

The application of *MeDMoT* to the two cases studies is presented in Sect. 6.1 and 6.2 respectively.

6.1 U2OWL: a Transformation from UML Models to OWL

In this section we describe the application of *MeDMoT* to develop a transformation from UML models of ontologies (model written as described in Sect. 4.3) into ontologies written using the RDF/XML concrete syntax of OWL. We have not yet implemented this transformation.

6.1.1 Transformation Requirements

MeDMoT prescribes that to specify the requirements, we must define the following artifacts.

[1]www.hibernate.org/

Source the source of the U2OWL transformation are the UML models of ontologies as described in Sect. 4.3.

Target the target of the transformation is a document describing an ontology written using OWL, and precisely the RDF/XML [2] for OWL concrete syntax. RDF/XML for OWL RDF/XML with a particular translations for OWL constructs[3] .

Characterization of Transformation Relation in the U2OWL case we have that the transformation must produce an OWL definition of an ontology having exactly the categories and the instances described by the input UML model together with their features (e.g., attributes, relationships, slots) as defined by that UML model.

A class with the stereotype ≪category≫ contained in the *Static View* corresponds to a class description with the same name of the UML class. An attribute of a class corresponds to a datatype property having as domain the name of the owner class, and as name the name of the attribute. Moreover, the class definition must contains a property restriction linked with the datatype property by means of the onProperty element.

Each directed association between two classes corresponds to an object property named with the name of the ending role of the association, and to a property restriction contained in the class description that corresponds to the UML class from which the association start.

A UML object corresponds to a OWL individual having a data property assertion for each slot of the UML object, and having an object property assertion for each link starting from that UML object.

6.1.2 Transformation Design

Well-Formedness Check

In this case there is no need of pragmatical or implementation dependant well-formedness rules to add to those listed in Sect. 4.3. Since the definition of the diagnostic metamodel in this case does not present any particularities, it can be done in the standard way showed in Sect. 5.3.1. Thus, in this case we do not show the diagnostic metamodel.

Refactoring

As suggested by *MeDMoT* our UML ontology models are refactored using one or more model-to-model transformations to keep the subsequent model to text transformation as simple as possible (see Sect. 5.3.2).

[2]`www.w3.org/TR/owl2-primer/`
[3]OWL 2 RDF Mapping, `www.w3.org/TR/owl2-mapping-to-rdf/`

In the U2OWL case there are two refactorings:

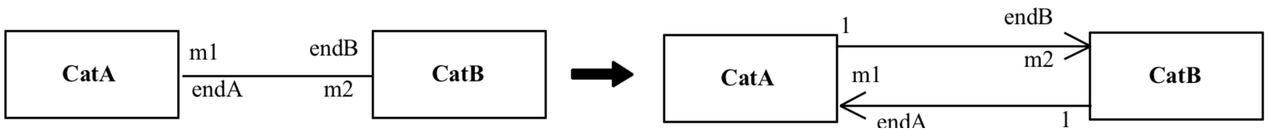
1. bidirectional associations between categories are allowed by the well-formdness rules (see Sect. 4.3), this refactor transform each bidirectional association into two oriented associations, thus the following TGT has only to cope with this kind of associations;

2. ≪instances≫ entities model at the same time individuals and classes, expanding the classes stereotyped by ≪instances≫ into a class belonging to the same category and a set of objects, the following TGT can avoid to cope with classes stereotyped by ≪instances≫.

- **RemoveBidirectionalAssociations**

It transforms a bidirectional association into two oriented associations. The multiplicities of the ending roles are preserved.

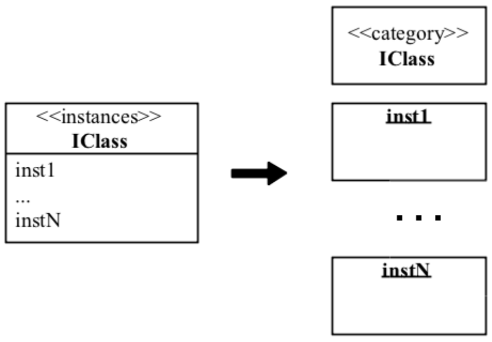
Refactoring case



- **Expand ≪instances≫**

It expands a class IS stereotyped by ≪instances≫ into the same class stereotyped by ≪category≫ and into a set of objects to be added to the Instance View.

Refactoring case



Design of the Text Generation Transformation

Design of transformation Output

The considered UML models of ontologies (see Sect. 4.3) contain classes with the stereotype ≪category≫ that represent classes in the ontology. They may be realized by means OWL *classes*. For example, a UML class Human will be realized by:

```
 <owl:Class rdf:ID='Human' />
```

The generalization relationship between classes has the same semantic meaning of *subClassOf* part of the RDF schema, so for example, the UML class stereotyped by ≪category≫ C specializing A, will be transformed in:

```
<owl:Class rdf:ID='C'>
 <rdf:subClassOf rdf:resource='#A'/>
</owl:Class>
```

The attributes of a UML class can be mapped to *datatype properties* in OWL, or to *object properties* depending on the type of the attributes. If the type of the attribute is a UML primitive type, then the attribute can be mapped to a *datatype property*, otherwise it can be mapped to a *object properties*. Given that in our domain definition the well-formedness rules require that attributes must be typed only with UML primitive types, we represent attributes with *datatype property*. For example the attribute attr of type int owned by a UML class named C will be realized by:

```
<owl:dataTypeProperty rdfID='attr'>
  <rdfs:domain rdf:resource='#C'/>
  <rdfs:range rdf:resource=''http://www.w3.org/2001/XMLSchema#integer'
     '/>
</owl:dataTypeProperty>
```

The refactoring phase guarantees that we have only associations navigable in one direction, moreover the well-formedness constraints on the source models guarantee that all the associations have named ends and are anonymous, thus we can identify the *object property* with the name of the association end. For example, an association between class C and class C1 with association end named endC1, will be realized by:

```
<owl:objectProperty rdfID='endC1'>
  <rdfs:domain rdf:resource='#C'/>
  <rdfs:range rdf:resource='#C1'/>
</owl:objectProperty>
```

Moreover, our well-formedness rules prescribe that all the attributes must have a multiplicity equal to one. Thus we can map the attribute cardinalities to OWL *restriction* and *subClassOf* between the OWL *class* and the OWL *restriction*. Thus, the restriction over the attr used in example before, will be realized by:

```
<rdfs:subClassOf>
  <owl:Restriction>
    <owl:onProperty rdf:resource='#attr'/>
    <owl:cardinality rdf:datatype = ``http://www.w3.org/2001/XMLSchema
       #integer''/>
  </owl:Restriction>
</rdfs:subClassOf>
```

In a similar way, we can realize the cardinality of an association end by means of OWL *minCardinality* and *maxCardinality*. Thus, the definition of the cardinality (from inf to sup) of the association end endC1 used before, will be realized by:

```
<rdfs:subClassOf>
 <owl:Restriction>
  <owl:onProperty rdf:resource='#endC1'/>
  <owl:minCardinality rdf:datatype=``http://www.w3.org/2001/XMLSchema#
     nonNegativeInteger''>
    inf
  </owl:minCardinality>
        <owl:maxCardinality rdf:datatype=``http://www.w3.org/2001/
           XMLSchema#nonNegativeInteger''>
    sup
        </owl:maxCardinality>
 </owl:Restriction>
</rdfs:subClassOf>
```

UML objects have the same semantic meaning of individuals, thus they can be mapped to OWL individuals. For example, an object ind instance of C can be represented as follows:

```
<C rdf:ID='ind'/>
```

UML slots are single attributes or features owned by an instance specification (like an object) that include a value. In OWL there are data property assertions whose meaning is comparable. Indeed, a data property assertion states that the value of a data property for an object denoted by a given individual is a given constant. To enforce the univocity of data property assertion we chose to use as name of the data property assertion the name of the UML attribute (though, the well formedness-rules states that two attributes of two different classes must have different names). For example, the slot for the attribute attr1 with the value 3, may be specified inserting the following definition of an individual:

```
<attr1 rdf:datatype=``http://www.w3.org/2001/XMLSchema#int''>
  3
</attr1>
```

Analogously, we can map the UML links (instances of associations) into OWL object property assertions. For example, a link between an object instance of a class C, to another object whose name is objectName can be represented as follows.

```
<endC1 rdf:resource='objectName'/>
```

Text Generation Transformation Architecture

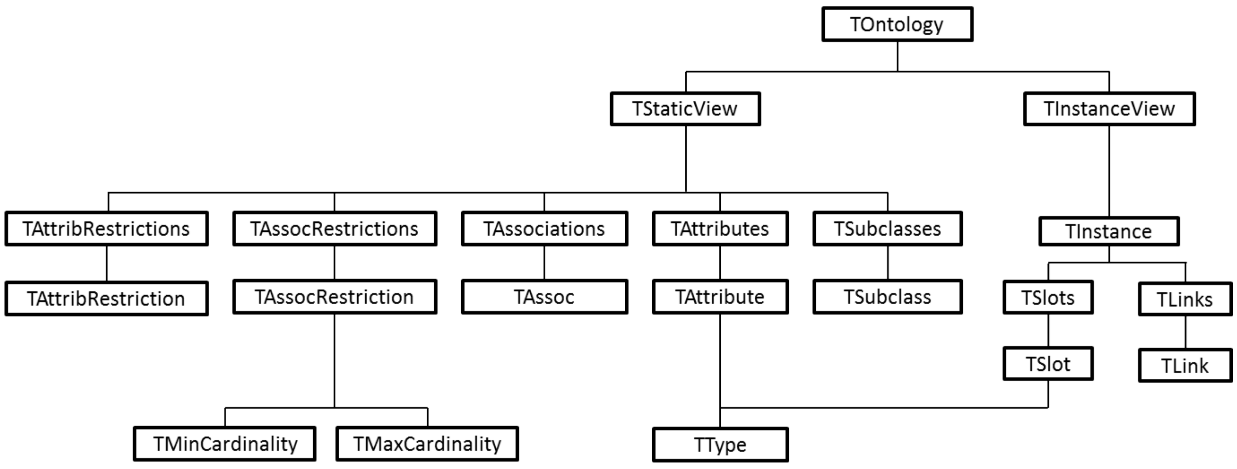


Figure 6.1: U2OWL: decomposition diagram

The structure of the text generation component of U2OWL is shown in Fig. 6.1.

Text Generation Transformation Design

In this case we present the TGT in both *User Level Design* and *Developer Level Design*.

User Level Design

The functions defining the Text Generation Transformation are given in the following.

- **TOntology**(ClassDiagram,ObjectDiagram)

It transforms an ontology model into an ontology expressed using the RDF/XML for OWL concrete syntax.

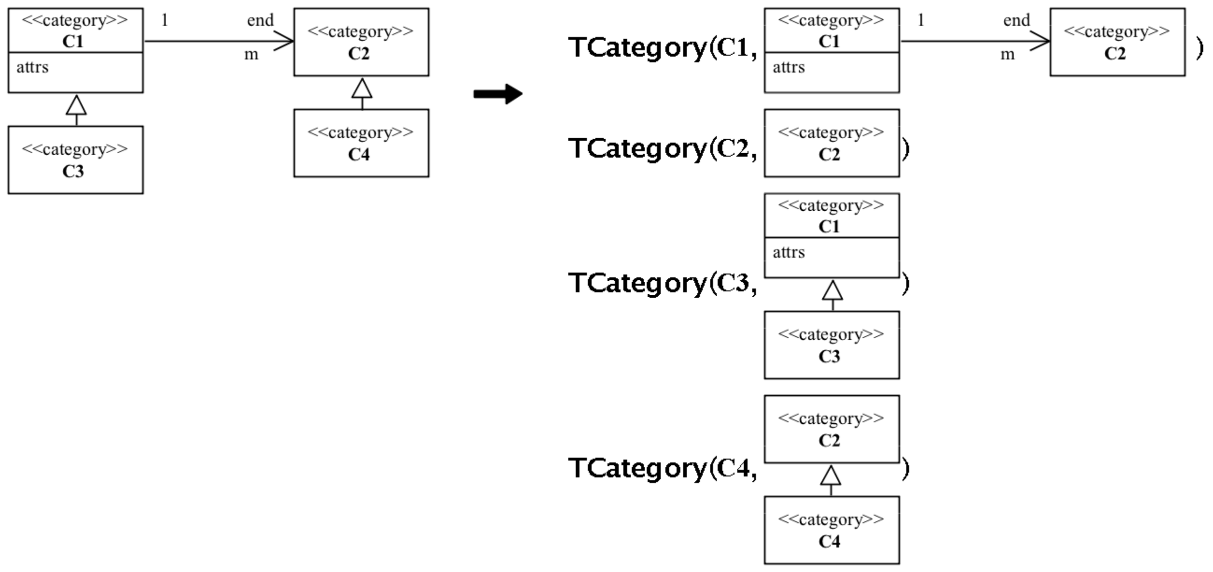
Definition

```
TOntology(SV, IV) =
<?xml version = '1.0' encoding = 'ISO-8859-1' ?>
<rdfRDF xmlns:rdf = '
http://www.w3.org/1999/02/22-rdf-syntax-ns#'
xmlns:rdfs = 'http://www.w3.org/2000/01/rdf-schema#'
xmlns:xsd = 'http://www.w3.org/2001/XMLSchema#'
xmlns:owl = 'http://www.w3.org/2002/07/owl#'
xmlns = 'http://dibris.org/Ontology #'
xml:base = 'http://dibiris.org/Ontology'>
<owl:ontology rdf:about="Ontology"/>
TStaticView(SV)
TInstancesView(IV)
</rdf:RDF>
```

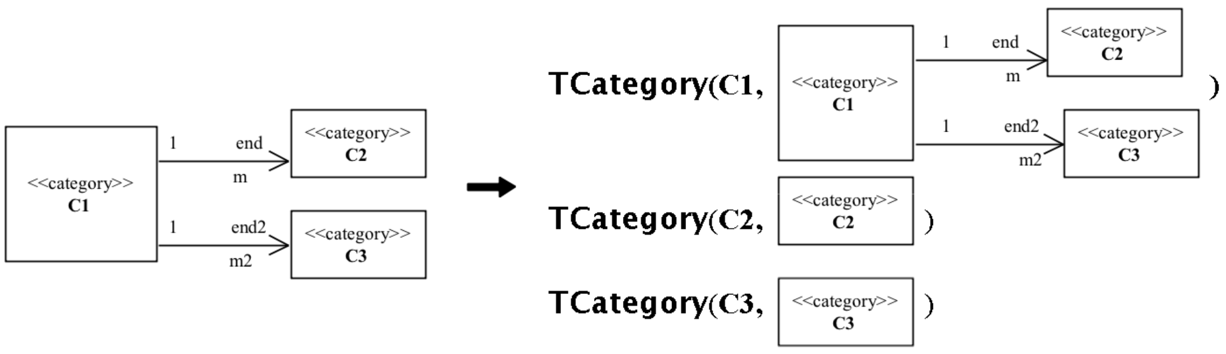
- **TStaticView**(ClassDiagram)

It transforms the static view into the corresponding ontology.

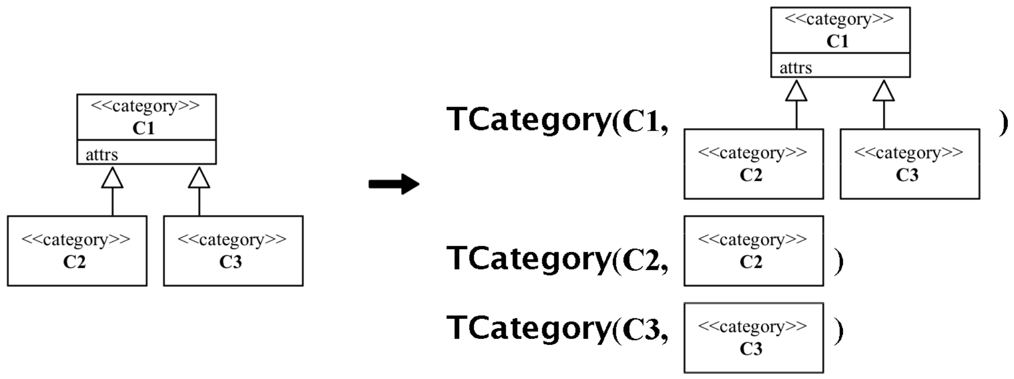
Transformation case



Transformation case



Transformation case

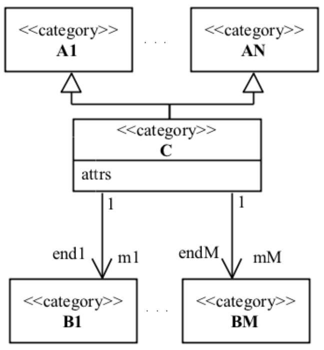


- **TCategory**(String,ClassDiagram)

It transforms a category with its attributes, associations and specializations into OWL classes, data properties, object properties and the needed restrictions over them. The first parameter is the name of the category.

Transformation case

C

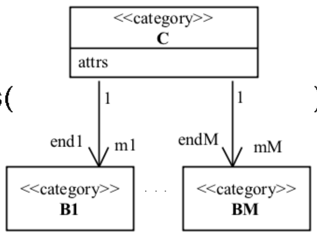


➡

<owl:Class rdf:ID="C">

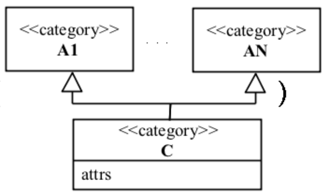
TAttributeRestrictions(attrs)

TAssocRestrictions(



)

TSubclasses(

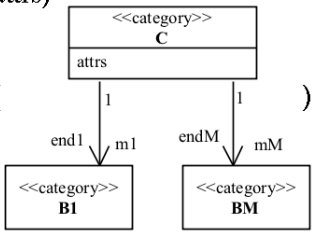


)

</owl:Class>

TAttributes(C, attrs)

TAssociations(



)

- **TAttribRestrictions**(Sequence(Property))

It transforms a sequence of category attributes into as many subsumption relations and owl restrictions as the corresponding data properties.

Definition

TAttributeRestrictions(attr1 ... attrN) =
    TAttributeRestriction(attr1) ... **TAttributeRestriction**(attrN)

- **TAttribRestriction**(Property)

Given a category attribute returns a subsumption relation and OWL restriction specifying a cardinality restriction over the corresponding data property equal to one (in the source the category attributes have always multiplicity equal to 1).

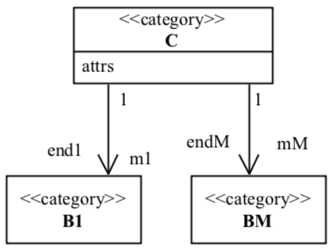
Definition

```
TAttributeRestriction(id: type) =
<rdfs:subClassOf>
   <owl:Restriction>
      <owl:onProperty rdf: resource = '#id'/>
         <owl:cardinality rdf:datatype = "http:www.w3.org/2001/XmlSchema#nonnegativeInteger">
            1
         </owl:cardinality>
   </owl:Restriction>
</rdfs:subClassOf>
```

- **TAssocRestrictions**(ClassDiagram)

It transforms a class diagram containing a category and M categories associated with it into M subsumption relations and OWL restrictions over the object properties representing M the associations.

Definition



$\Rightarrow$

TAssocRestriction(end1, m1)
...
TAssocRestriction(endM, mM)

- **TAssocRestriction**(String,Multiplicity)

It transforms the multiplicity of the ending role of an association into a subsumption relation and an OWL restriction over the object property representing the association. The association end name and the corresponding multiplicity are given as parameters.

Definition

```
TAssocRestriction(end, inf..sup) =
   <rdfs:subClassOf>
   <owl:Restriction>
   <owl:onPropertyrdf:resource='end'/>
      TminCardinality(inf)
      TmaxCardinality(sup)
   </owl:Restriction>
   </rdfs:subClassOf>
```

- **TMinCardinality**(String)

It transforms the lower bound of the association end multiplicity.

Definition

```
TminCardinality(inf) =
   <owl:minCardinality rdf:datatype=  "http://www.w3.org/2001/XMLSchema#nonNegativeInteger">
      inf
   </owl:minCardinality>
```

- **TMaxCardinality**(String)

It transforms the upper bound of the association end multiplicity. If the upper bound is "" there's no need to restrict the upper bound of the object property representing the association.*

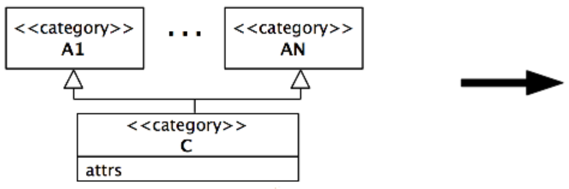
Definition

```
TMaxCardinality(sup) =
if sup ≠ "*" then
   <owl:maxCardinality rdf:datatype= "http://www.w3.org/2001/XMLSchema#nonNegativeInteger">
      sup
   </owl:maxCardinality>
else ""
```

- **TSubclasses**(ClassDiagram)

It transforms a class diagram containing a category and N categories from which it subclass, into N subsumption relations in owl.

Definition



TSubclass(A1)
. . .
TSubclass(AN)

- **TSubclass**(String)

It transforms a specialization relation into a subsumption relation in owl.

Definition

```
TSubclass(A)=
<rdf:subClassOf rdf:resource='#A'/>
```

- **TAttributes**(String,Sequence(Property))

It transforms a sequence of category attributes into a sequence of data properties.

Definition

```
TAttributes(attrs)=
                    TAttribute(C,attr1)
                     ...
                     TAttribute(C,attrQ)
```

- **TAttribute**(String,Property)

It transforms a category attribute into a data property. The domain axiom is qualified with the attribute name and the range axiom is the transformation of the attribute type. The category to which the attribute belongs is given as first parameter.

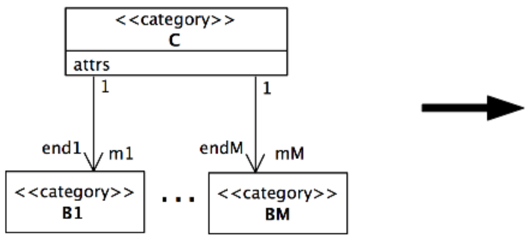
Definition

```
TAttribute(C, id: type) =
<owl:dataTypeProperty rdf:ID='id'>
   <rdfs:domain rdf:resource='#C'>
   <rdfs:range rdf:resource=TType(type)>
</owl:dataTypeProperty>
```

- **TAssociations**(ClassDiagram)

It transforms a class diagram containing a category and M categories associated with it into M object properties.

Definition



```
TAssoc(C, end1,B1 )
...
TAssoc(C,endM,BM )
```

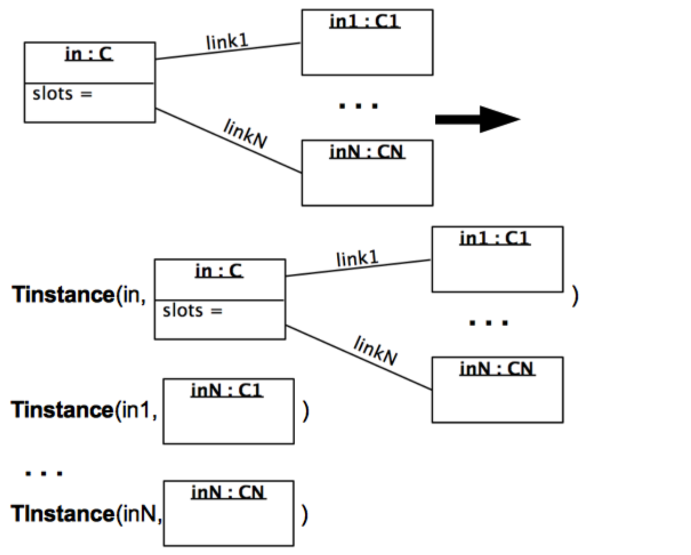
- **TAssoc**(String,String,String)
Given a category, the name of the end of an association starting from it into another category, and the name of the target category of such association, it returns an object property whose id is the end name. The domain axiom is qualified with the category name; the resource axiom is qualified with the name of the category to which the association is directed.

Definition

```
TAssoc(C1,endN,C2) = <owl:objectProperty rdf:ID= 'endN'>
                       <rdf:domain rdf:resource= '#C1'/>
                       <rdf:range rdf:resource= '#C2'/>
                      </owl:objectProperty
```

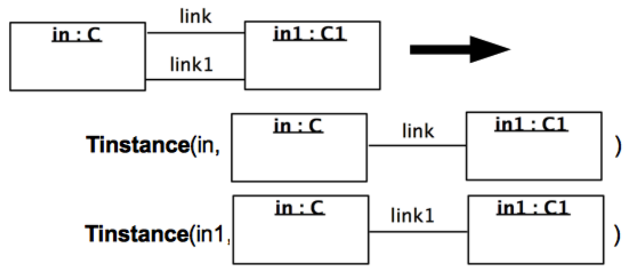
- **TInstancesView**(ObjectDiagram)
It transforms the objects and links contained in the Instances View into the corresponding individuals together with their properties.

Transformation case



where link1, ..., linkN are links of oriented association starting from C

Transformation case

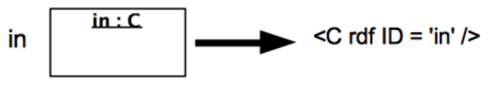


where link is a link of an oriented association from C to C1, and link1 of an oriented association from C1 to C

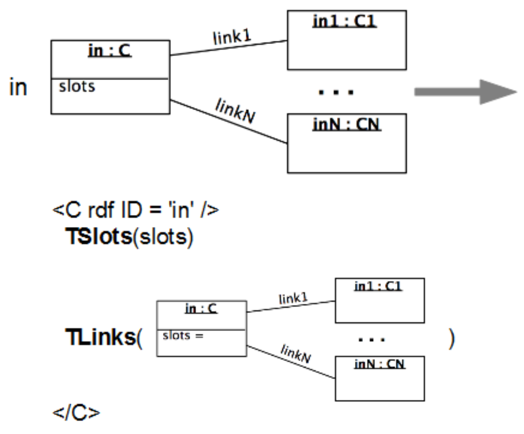
- **TInstance**(String,ObjectDiagram)

It transform an object into an individual having as type the class of the object and as name the name of the object. Slots and links of the object are transformed into data property assertions and object property assertions respectively. The first parameter is the name of the individual.

Transformation case



Transformation case



- **TSlots**(Sequence(AttributeInstance))

It transforms a sequence of slots into OWL data property assertions.

Definition

```
TSlots(slots) = TSlot(slotName1,slotType1,slotValue1)
                ...
                Tslot(slotNameQ,slotTypeQ,slotValueQ)
```

where for i=1,...,Q slotNamei is the name of the slot, slotTypei is the type slot and slotValue1 is the value of the slot.

- **TSlot**(String,Type,String)

It transforms a slot into an OWL data property assertion.

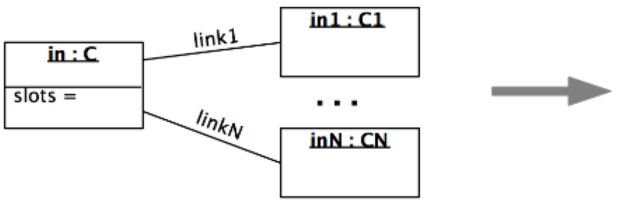
Definition

```
TSlot(attr,type,value) = <attr rdf:datype = TType(type)
                          value
                         </attr>
```

- **TLinks**(ObjectDiagram)

It transforms a sequence of links into object property assertions.

Definition



```
TLink(end1,in1)
...
TLink(endN,inN)
```

where for i=1,...,N endi is the ending role of the oriented association from C to Ci typing linki

- **TLink**(String,String)

It transforms a link (instance of an association) into an object property assertion.

Definition

```
TLink(assEnd,inst) = <assEnd rdf:resource = 'instance' />
```

Developer Level Design

As we have already done for the U2SQL case (see Sect. 5.3) we give the transformation design at developer-level.

Also in this case, the transformation is defined by giving the conceptual metamodel of the source, and the definitions of model to text functions.

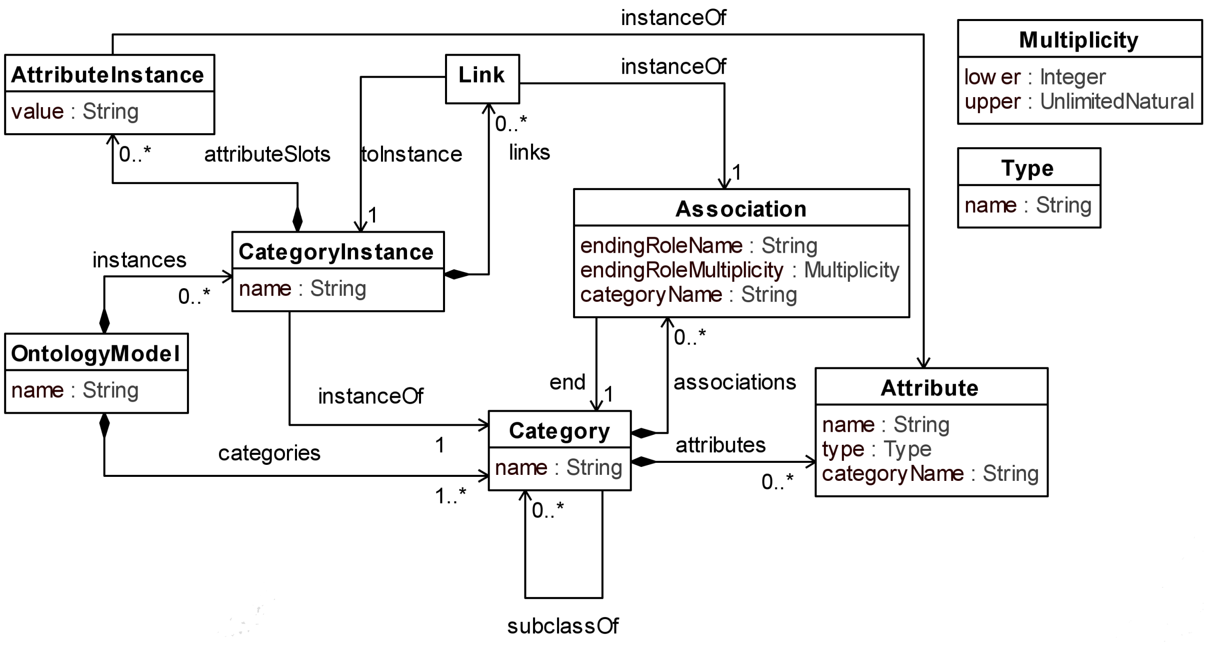


Figure 6.2: Ontology metamodel Developer Level Design

Fig. 6.2 shows the conceptual metamodel at *Developer Level Design*.

The following is a brief explanation of each element of the conceptual metamodel highlighting the UML metamodel element to which it has a relation.

Element Name	**Description**	**Related UML Meta-model element**
OntologyModel	An ontology	UML::Model
StaticView	A class diagram representing the structure of an ontology	UML::Package. May be the same package of the Ontology
InstanceView	An object diagram representing individuals	UML::Package. May be the same package of the StaticView
Category	A class stereotyped ≪category≫	UML::Class
Attribute	An attribute of a category	UML::Property
Association	A binary association between two categories	UML::Association
Multiplicity	A multiplicity used to define association cardinalities	UML::MultiplicityElement
Type	A type allowed for attributes	UML::PrimitiveType
CategoryInstance	An instance of a category	UML::InstanceSpecification
AttributeInstance	A slot for attributes of a category	UML::InstanceSpecification
Link	An instance of an association	UML::InstanceSpecification

The functions defining the transformation are given in the following.

- **TOntology**(OntologyModel)

It transforms an ontology model into an OWL ontology.

Definition

TOntology(*oM*) =
<?xml version = '1.0' encoding = 'ISO-8859-1' ?>
  <rdf:RDF xmlns:rdf = 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'
   xmlns:rdfs = 'http://www.w3.org/2000/01/rdf-schema#'
   xmlns:xsd = 'http://www.w3.org/2001/XMLSchema#'
   xmlns:owl = 'http://www.w3.org/2002/07/owl#'
   xmlns = 'http://dibris.org/OM.name'
   xml:base = 'http://dibiris.org/OM.name >
   <owl:ontology rdf:about="OM.name"/>
   TStaticView(*oM*.categories)
   TInstancesView(*oM*.instances)
</rdf:RDF>

- **TStaticView**(Sequence(Category))

It transforms a sequence of categories each category with its attributes, associations and specialization relations into and OWL classes, data properties, object properties and the needed *restrictions over these properties.*

Definition

TStaticView(*category*$_1$, ..., *category*$_n$) = **TCategory**(*category*$_1$), ..., **TCategory**(*category*$_n$)

- **TCategory**(Category)

It transforms a category with its attributes, associations and specialization relations into and OWL class, data properties, object properties and the needed restrictions over these properties.

Definition

TCategory(*category*) =
<owl:Class rdf:ID=' "#" + *category*.name >
  TAttribRestrictions(*category*.attributes)
  TAssocRestrictions(*category*.associations)
  TSubclasses(*category*.subclassOf)
</owl:Class>
TAttributes(*category*.attributes)
TAssociations(*category*.associations)

- **TAttribRestrictions**(Sequence(Attribute))

It transforms a sequence of category attributes into as many subsubtion relations and owl restrictions as the corresponding data properties.

Definition

TAttribRestrictions(*attr*$_1$, ..., *attr*$_n$) =
TAttribRestriction(*attr*$_1$), ..., **TAttribRestriction**(*attr*$_n$)

- **TAttribRestriction**(Attribute)

Given a category attribute returns a subsumption relation and OWL restriction specifying a cardinality restriction over the corresponding data property equal to one (in the source the category attributes have always multiplicity equal to one).

Definition

TAttribRestriction(*attr*) =
<rdfs:subClassOf>
  <owl:Restriction>
  <owl:onProperty rdf:resource = ' "#" + *attr*'/>

<owl:cardinality rdf:datatype =
"http://www.w3.org/2001/XMLSchema#nonNegativeInteger" >
1
</owl:cardinality >
</owl:Restriction>
</rdfs:subClassOf>

- **TAssocRestrictions**(Sequence(Association))

It transform a sequence of associations into subsumption relations and an OWL restrictions over the object properties representing the associations.

Definition

TAssocRestrictions($assoc_1, \ldots, assoc_n$) =

TAssocRestriction($assoc_1$), …, **TAssocRestriction**($assoc_n$)

- **TAssocRestriction**(Association)

It transforms the multiplicity of the ending role of an association into a subsumption relation and an OWL restriction over the object property representing the association.

Definition

TAssocRestriction(*assoc*) =
<rdfs:subClassOf>
<owl:Restriction>
<owl:onProperty rdf:resource='#*assoc*.endRoleName'/>
TMinCardinality(*assoc*.endingRoleMultiplicity.lower)
TMaxCardinality(*assoc*.endingRoleMultiplicity.upper)
</owl:Restriction>
</rdfs:subClassOf>

- **TMinCardinality**(String)

It transforms the lower bound of the association end multiplicity.

Definition

TMinCardinality(*lower*) =
<owl:minCardinality rdf:datatype=
"http://www.w3.org/2001/XMLSchema#nonNegativeInteger">
lower

</owl:minCardinality>

- **TMaxCardinality**(String)

It transforms the upper bound of the association end multiplicity. If the upper bound is "" there is no need to restrict the upper bound of the object property representing the association.*

Definition

TMaxCardinality(*upper*) =
if *upper*<> "*" then
    <owl:maxCardinality rdf:datatype=
    "http://www.w3.org/2001/XMLSchema#nonNegativeInteger">
    upper
</owl:maxCardinality>
else ""

- **TSubclasses**(Sequence(Category))

It transforms a sequence of specialization relations.

Definition

TSubclasses($category_1$, . . . , $category_n$) = **TSubclass**($category_1$), . . . , **TSubclass**($category_n$)

- **TSubclass**(Category)

It transforms a specialization relation into a subsumption relation in owl.

Definition

TSubclass(*category*) = <rdf:subClassOf rdf:resource="'#"+ *category*.name/>

- **TAttributes**(Sequence(Attribute))

It transforms a sequence of category attributes into a sequence of data properties.

Definition

TAttributes($attr_1$, . . . , $attr_n$) =

TAttribute($attr_1$), . . . , **TAttribute**($attr_n$)

- **TAttribute**(Attribute))

It transforms a category attribute into a data property. The domain axiom is qualified with the *attribute's name and the range axiom is the transformation of the attribute's primitive type (due to well-formedness rules category's attribute can have only UML primitive types) by means of the **TType** function.*

Definition

TAttribute(*attr*) =
<owl:DataTypeProperty rdf:ID='*attr*.name'>
    <rdfs:Domain rdf:resource=' "#" + *attr*.categoryName'/>
    <rdfs:range rdf:resource= **TType**(*attr*.type)/>
</owl: DataTypeProperty >

- **TAssociations**(Sequence(Association))

It transforms a sequence of associations into object properties.

Definition

TAssociations($assoc_1$, ..., $assoc_n$) = **TAssoc**($assoc_1$), ..., **TAssoc**($assoc_n$)

- **TAssoc**(Association)

Given a category, the name of the end of an association starting from it into another category, and the name of the target category of such association, it returns an object property whose id is the end name. The domain axiom is qualified with the category name; the resource axiom is qualified with the name of the category to which the association is directed.

Definition

TAssoc(*assoc*) =
<owl:objectProperty rdf:ID=*assoc*.endingRoleName'>
    <rdf:domain rdf:resource=' "#" +*assoc*.categoryName'/>
    <rdf:range rdf:resource=' "#" + *assoc*.end.name'/>
</owl:objectProperty >

- **TInstancesView**(Sequence(CategoryInstance))

It transforms a sequence of objects into the corresponding individuals together with their properties.

Definition

TInstancesView($object_1$, ..., $object_n$) =

TInstance($object_1$), ..., **TInstance**($object_n$)

- **TInstance**(CategoryInstance)

It transform an object into an individual having as type the class of the object and as name the name of the object. Slots and links of the object are transformed into data property assertions and object property assertions respectively.

Definition

TInstance(*object*) =
<*object*.instanceOf.name rdf:ID='*object*.name '>
  TSlots(*object*.attributeSlots)
  TLinks(*object*.links)
</*object*.instanceOf.name>

- **TSlots**(Sequence(AttributeInstance))

It transforms a sequence of slots into OWL data property assertions.

Definition

TSlots($slot_1, \ldots, slot_n$) = **TSlot**($slot_1$), ..., **TSlot**($slot_n$)

- **TSlot**(AttributeInstance)

It transforms a slot (instance of a class's attribute) into an OWL data property assertion.

Definition

TSlot(*slot*) =
<*slot*.instanceOf.name
  rdf:datatype= **TType**(*slot*.type)>
  slot.value
</*slot*.instanceOf.name>

- **TLinks**(Sequence(Link))

It transforms a sequence of links into object property assertions.

Definition

TLinks($link_1, \ldots, link_n$) = **TLink**($link_1$), ..., **TLink**($link_n$)

- **TLink**(Link)

It transforms a link (instance of an association) into an object property assertion.

Definition

TLink(*link*) = <*link*.instanceOf.endingRoleName rdf:resource='*link*.toInstance.name'/>

6.1.3 Transformation Testing

The test suite designed following our method as described in Sect. 5.4 are reported in Appendix A.

6.2 U2Java: a Transformation from MJavaDModels to Java Desktop Applications

U2Java transforms models written applying the method MJavaD(see Sect. 4.4) to Java desktop applications (excluding the GUI). The implementation of this transformation lead us to implement the AutoMARS tool, see Sect. 6.2.4.

6.2.1 Transformation Requirements

Source the source of the U2Java transformation are exactly the models defined by MJavaD, in this case no new well-definedness rules have been added nor any existing one has been removed.

Target The target of the transformation are projects, complete of source code and configurations files, of Java desktop applications. The code composing the projects is Java 7, using the *Spring* framework and the Java Persistent API (JPA) with *Hibernate* as persistence provider, to manage object persistence. Moreover, these projects are manageable with *Maven*.

Maven [2, 92] is a project management tool that provides a building infrastructure. It is a tool that, based on a common description of the project (the Project Object Model or shortly POM), can compile, test, report and document a Java project. Each project built with *Maven* has a unique identifier, thus the dependency (libraries and frameworks) can be declared and used by *Maven* to get the right packaged bytecode (also from internet sites). Each of the tasks that can be done with *Maven* (called "goals"), when activated by the user, causes the execution of the sequence of steps (e.g., compile, execute some tests and if they give positive answers, package the bytecode into the right way) needed to accomplish that task.

Spring [29] is an open source framework and lightweight container for Java (enterprise) applications. It offers an infrastructure to manage simple Java Objects (called Plain Old Java Objects, POJO) to collaborate each other to form the application. It is composed by about twenty modules, each of which provides a part of the framework features. *Spring* offers a base set of features (dependency injection[4] features and an implementation of the factory patter) plus others for specific parts of the application, like data access, web, AOP[5] instrumentation and test.

[4]see `martinfowler.com/articles/injection.html` for a brief explanation of what dependency injection is.

[5]Aspect Oriented Programming

Java Persistence Application Programming Interface (API) [14] is a Java API specification which give a description of how to manage POJO persistence using object-relational mapping (ORM)[6]. It was developed to be used in enterprise Java but can be used also non enterprise Java Applications. The *Spring* framework supports JPA.

Hibernate [13] is an ORM implementation which provides high level functions to manage objects persistence in an object oriented way. It is also an implementation of the JPA specification.

Characterization of Transformation Relation Most of the semantic concepts present in the source models have a "natural" correspondence into the semantic of Java (e.g. the generalization and the association relations). Thus, we do not give a detailed list of these correspondences. Here, we give a description of the correspondences among the entities described in Sect. 4.4 and elements belong to the transformation target, (elements that may be build using Java and the frameworks/libraries composing the *Target*).

Active classes, like classes with the stereotype ≪executor≫ or ≪boundary≫, correspond to asynchronous *Spring* managed beans (they are Java threads managed by *Spring*).

Classes with the stereotype ≪store≫ or ≪singletonStore≫ correspond to JPA [14] entity classes.

Classes with the stereotype ≪context≫ correspond to Java classes able to post modification request to a Swing[7] GUI (not generated by the transformation).

6.2.2 Transformation Design

Well-Formedness Check

In this case there is no need of pragmatical or implementation dependant well-formedness rules to add to those listed in Sect. 4.4.

Refactoring

In the following, we give a description of each refactoring used in this case. They must be executed in the same order in which they are described.

1. The unnamed associations are to be removed, since they only helps to understand the dependency among the various classes composing the design of the application;

[6]ORM is a way to overcome the differences between the object model and the relational model.

[7]`docs.oracle.com/javase/tutorial/uiswing/start/about.html`

2. for each class with a ≪create≫ operation (all the classes with the stereotypes ≪store≫, ≪singletonStore≫, ≪executor≫,≪boundary≫ and the datatypes must have a ≪create≫ operation, see Sect. 4.4) a constructor with the same parameter of the ≪create≫ operation is added, and the method associated to the ≪create≫ operation is de-associated from that operation and associated with the newly added Java constructor. This refactoring simplify the handling of invariant constraints in the templates of the following model to text transformation, indeed invariant must be checked also after the object creation;

3. a constructor without parameters is added to the classes stereotyped by ≪store≫ and ≪singletonStore≫, if they do not have it. This is required by the persistence framework used in target of the transformation;

4. all the classes with the stereotype ≪boundary≫ must be connected with the unique class stereotyped by ≪context≫. We have a design pattern for the boundary classes (boundary classes are classes with the ≪boundary≫ stereotype), so that there is one main boundary that orchestrate the other boundaries. This is represented associating the main boundary with the managed ones by means of an aggregation associations, and drawing a directed association from the main boundary to the context class (the class with the ≪context≫ stereotype). This refactoring allows the designer to do not draw the associations between the non-main boundaries and the context class, thus avoiding to dirtying the diagram with a lot of directed associations to the context class.

5. named associations must be removed and a new attribute typed by the class to which the association is oriented is inserted. This refactoring greatly simplifies the design of the following text generation transformation.

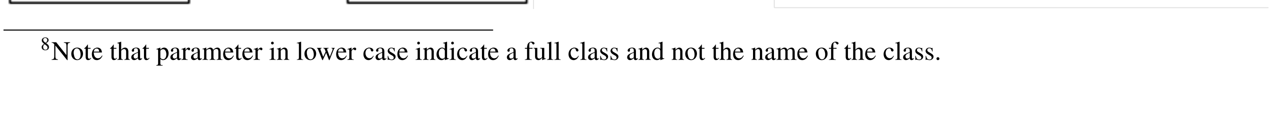
6. specializations are removed by duplicating attributes and associations of the superclasses in the subclasses. Also this refactoring greatly simplifies the design of the following text generation transformation.

Here there are the designs of some of these refactorings.

- **RemoveUnnamedAssociations**

It removes unnamed associations[8].

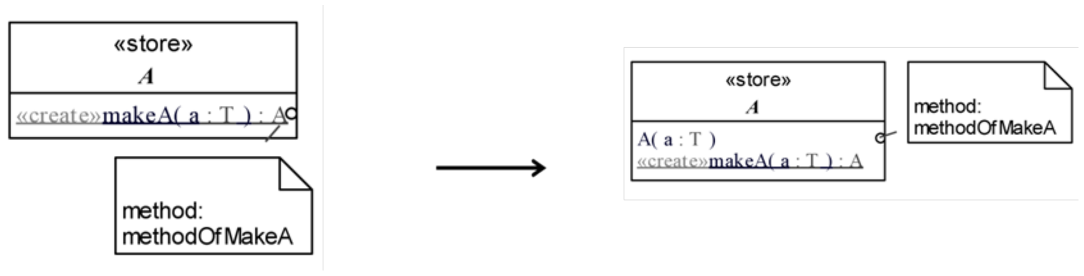
Refactoring case



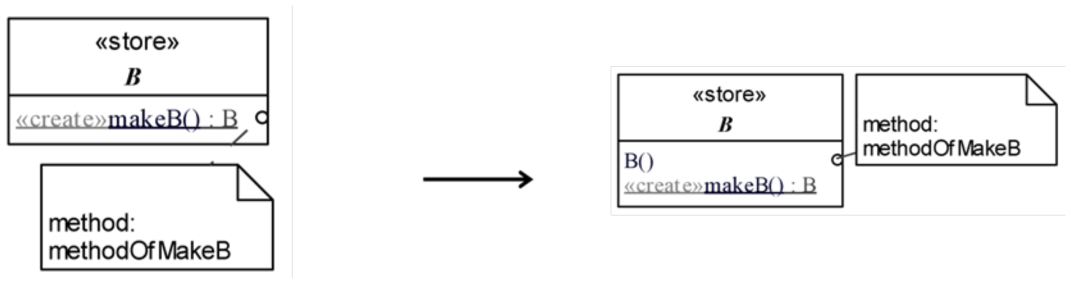
[8]Note that parameter in lower case indicate a full class and not the name of the class.

- **AddConstructorForCreateOp**
For each operation with the stereotype ≪create≫ it creates a constructor with the same parameters of the ≪create≫ operation. Moreover, it moves the method associated with the ≪create≫ operation to the newly created constructor[9]. *The* ***AddConstructorForCreateOp*** *refactoring is applied also to the classes with the stereotypes ≪singletonStore≫, ≪executor≫, ≪boundary≫ and to the datypes.*

Refactoring case



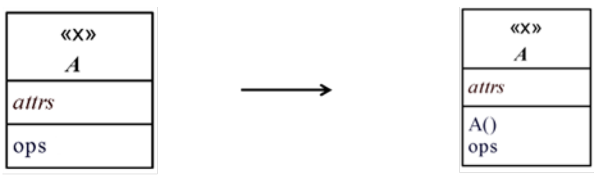
Refactoring case



[9]UML methods not visible in the class diagram are visualized using notes applied to the elements of the class diagram.

- **CreateDefaultConstructor**
For each class with the stereotype ≪store≫ or ≪singletonStore≫ it creates a default constructor. Default constructors are required by the persistence framework used in the implementation.

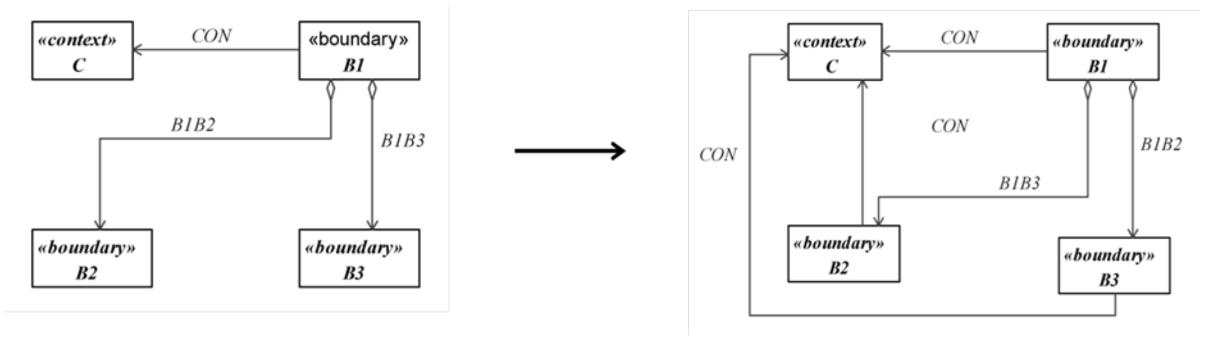
Refactoring case



Where x can be: store or singletonStore.

- **AddAggregatedBoundaryAssociation**
For each class stereotyped ≪boundary≫ that is aggregated (i.e., connected by an aggregation association with another class with the same stereotype) it creates a newly named associations between this class and the class stereotyped ≪context≫.

Refactoring case



U2Java: Design of Text Generation Transformation

Design of transformation Output The design of the Java desktop application result of the transformation is made of configurations and Java elements, plus frameworks elements. In the following we give a brief description of the design of the various parts that compose the Java desktop application, referencing the entities described in Sect. 4.4. The design descriptions are given at a rather high-level of abstraction, without mention low level implementation details.

Each entities used to build the application described in Sect. 4.4, must be designed using the elements of the transformation target. In the following we give a brief description of those designs.

Datatypes A *Datatype* is modelled as regular Java class with an appropriate *equals* method, plus standard getters and setters. Moreover, if in the source model the Datatype is used to type at least one attribute in a class with the stereotype ≪store≫ or ≪singletonStore≫, then the class, translation of the Datatype, is annotated with the @*Embeddable* annotation. The @*Embeddable*[10] annotation specify that instance of this class can be stored as a part of an owning entity.

Enums Enums are modelled as Java Enums.

Contexts A *Context* class is modelled as: (1) a Java interface containing all the method signatures corresponding to the *Context* operations and (2) a Java class which implements that Java interface. Moreover, the class must have an attribute typed by a marker interface (this attribute will be set to a reference to a Swing GUI). Each method of the class (2) will invoke a method with the same name in the Swing GUI by means of the *invokeLater*[11] method of the *SwingUtilities*[12] class.

Executors An *Executor* and the state machine associated with it are modelled as:

1. a Java interface containing all the methods signatures corresponding to the event operations of the *Executor*,
2. a Java enumeration containing all the states belonging to the state machine, and
3. a Java class containing the implementation of the associated state machine (implemented using nested switch [65]) and all the attributes and associations present in the *Executor*.

The Java class (3) is configured as a *Spring* managed-bean enabled for an asynchronous execution, thus class objects can execute in their own thread of execution. The event queue associated with the state machine is modelled as a Blocking Queue[13] of events. Each event is modelled as a Java class (*Event*) containing the name of the calling class, the name of the method associated with the event operation containing the implementation of the state transition and all the event parameters. Each call to one of the event operations of the *Executor* results in a creation of a new *Event* object and in the insertion of this newly created object in the Blocking Queue. During the thread execution *Event* objects are taken from the event queue and the information stored in these objects are used to call the appropriate class method. Operations with the stereotype ≪destroy≫ are modelled as special methods that may stop the thread execution.

[10] docs.oracle.com/javaee/7/api/javax/persistence/Embeddable.html

[11] ocs.oracle.com/javase/7/docs/api/javax/swing/SwingUtilities.html#invokeLater(java.lang.Runnable)

[12] docs.oracle.com/javase/7/docs/api/javax/swing/SwingUtilities.html

[13] see docs.oracle.com/javase/7/docs/api/java/util/concurrent/ArrayBlockingQueue.html

Directory	Description
src/main/java	Application sources
src/main/resources	Application resources
src/test/java	Test sources
src/test/resources	Test resources

Table 6.1: directory layout

Boundary A *Boundary* class is modelled in same way are modelled *Executor* classes.

Stores A *Store* class is modelled as: (1) a JPA entity class, (2) a Data Access Object (DAO) class to which data base operations are delegate. In JPA an entity class, is an annotated (annotation is a way to decorate JAVA source code with meta-data informations) Java class with the ability of representing objects in a databases and a DAO[14] (a class implementing the Data Access Object pattern) class to which CRUD[15] operations on entity are delegate. The DAO class has a reference to the entity manager[16] class and all the CRUD methods are annotated as transactional. *Store* attributes marked with the ≪key≫ stereotype are modelled as a Java class annotated with the @Embeddable annotation. Moreover, in the entity class has a reference to this class marked with the @EmbeddedId[17] annotation to denote a composite primary key that is an embeddable class.

Singleton stores A *Singleton Store* is modelled in the same way we have modelled the *Singleton*. The only difference is that there composite primary keys are not allowed and there is only an id field marked with @Id annotation[18] which has the value one. Moreover, the DAO class does not have the delete method (only save and update are allowed).

The Java Project

We decided to structure the output of the transformation as *Maven* Java project of a desktop application. *Maven* projects have a standard directory layout, thus the transformation output has the structure showed in Table 6.1.

The informations needed by *Maven* to build the project are contained in the Project Object Model (POM, see[19]), which is a XML file that contains informations about the project and configuration details, like project dependencies or required plugins (most of the *Maven* functionalities are in plugins) and their configuration.

Java projects produced by the transformation depend on the *Spring* framework, the *Hibernate*

[14] `www.oracle.com/technetwork/java/dataaccessobject-138824.html`
[15] CRUD stand for Create Read Update and Delete that are the four basic operations on a persistence storage
[16] `docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html`
[17] `docs.oracle.com/javaee/7/api/javax/persistence/EmbeddedId.html`
[18] `docs.oracle.com/javaee/7/api/javax/persistence/Id.html`
[19] `maven.apache.org/guides/introduction/introduction-to-the-pom.html`

libraries etc.. Thus, the POM produced by the transformation must contains dependency and configuration informations about those frameworks and libraries. The POM file must be contained in the project root folder.

In the following, we briefly describe the design of the other various parts composing the application.

OCL library The OCL types and collections are modelled using Java interfaces and classes related each other. Collection operations, like *forAll* or *exists*, are modelled, when required, emulating closures using Java anonymous inner classes.

Invariants Invariants are modelled in Java in the following way: an invariant is modelled with a Java method, say *inv*, which, if the invariant condition is evaluated to false, raise a specific type of exception. Calls to *inv* method is done in each method of each class, after each line of the method.

Pre-conditions and Post-conditions Operation pre-conditions, as well as post-conditions, are modelled as Java methods that can raise a specific type of exception if the condition (associated with the pre-postcondition) is evaluated to false.

U2Java: Text Generation Transformation Architecture and Design

Since in this case the size of the text generation transformation architecture is too large to show it in a single figure (as in other cases), we show parts of the decomposition diagram followed by the definition of the functions contained in the part showed. Moreover, in this case we show only a part of the transformation cases belonging to the *User Level Design*.

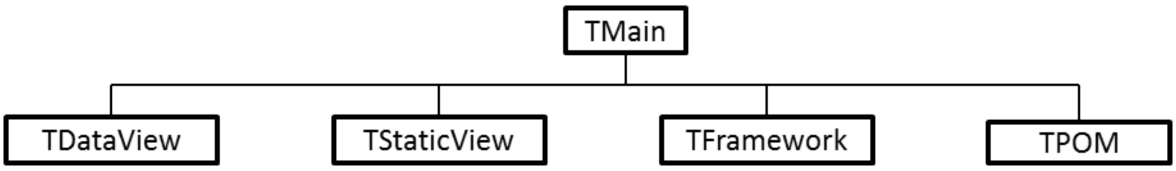


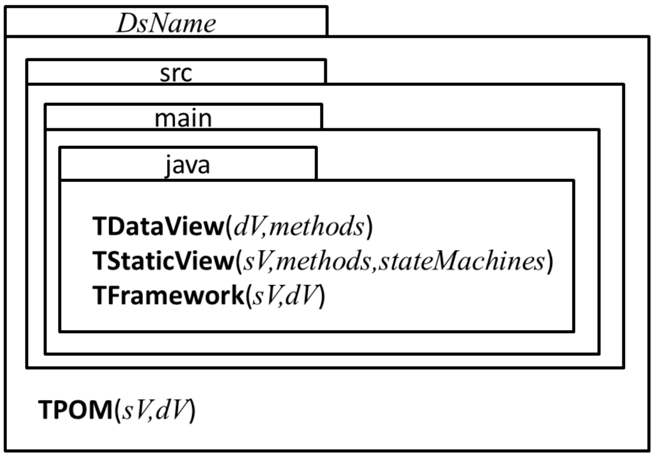
Figure 6.3: U2Java decomposition diagram: main transformation

- **TMain**(DesignSpecification)

It transforms a UML model of the design of a JAVA desktop applications into its implementation. The symbol at the left side represents a folder.

Definition

TMain(*DsName,sV,dV, methods,stateMachines*) =



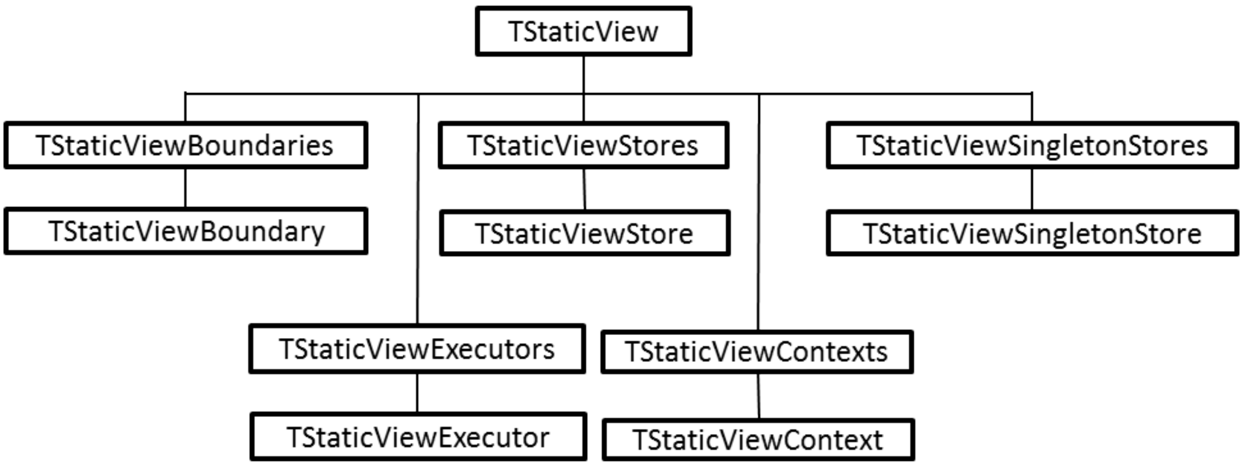


Figure 6.4: U2Java decomposition diagram: TStaticView

- **TStaticView**(StaticView)

It transforms the static view into the implementation of all the entities contained in it.

Definition

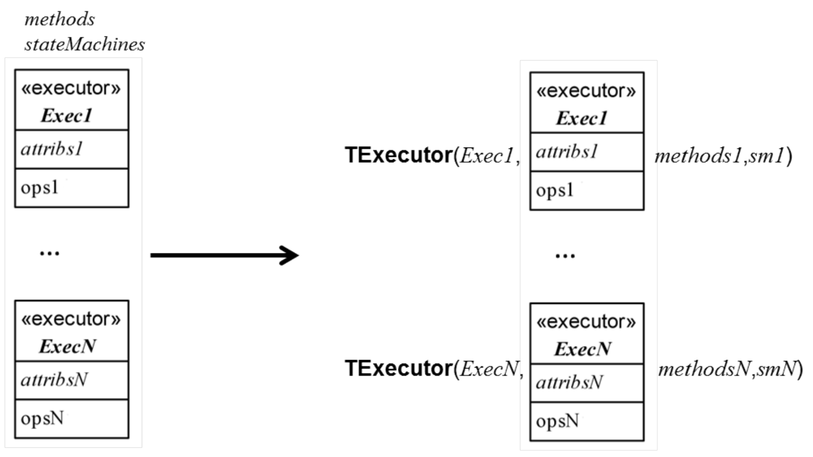
TStaticView(*sV,methods,StateMachines*) =
TStaticViewBoundaries(*boundaries,methods,StateMachines*)
TStaticViewExecutors(*executors,methods,StateMachines*)
TStaticViewStores(*stores,methods*)
TStaticViewSingletonStores(*singletonStores, methods*)
TStaticViewContexts(*contexts*)

where:	*boundaries*	are all the classes with the stereotype <<boundary>>,
	executors	are all the classes with the stereotype <<executor>>,
	stores	are all the classes with the stereotype <<store>>,
	singletonStores	are all the classes with the stereotype <<singletonStore>>,
	context	are all the classes with the stereotype <<context>>.

- **TStaticViewExecutors**(Sequence(String))

It transforms all the classes with the stereotype ≪executor≫ contained in the static view.

Transformation case

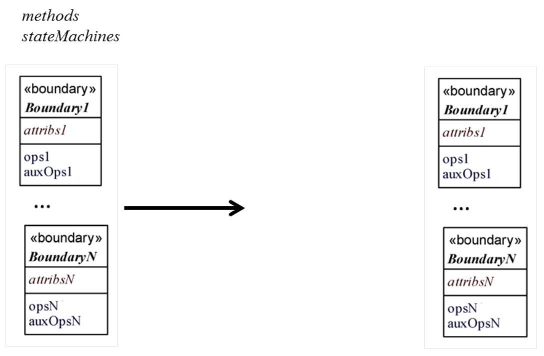


For i=1..N *methodsi* is the list of the methods of *Execi*,
methodsi belong to the *methods* list
For i=1..N *smi* is the state machine in the *stateMachines* list defining the behaviour of *Execi*

- **TStaticViewBoundaries**(Sequence(String))

It transforms all the classes with the stereotype ≪boundary≫ contained in the static view.

Transformation case

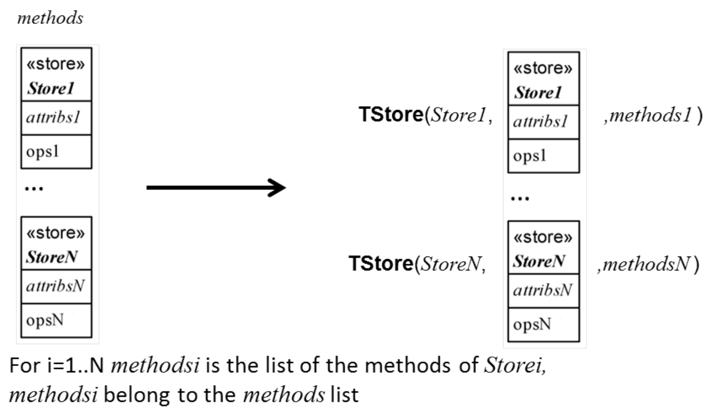


For i=1..N *methodsi* is the list of the methods of *Boundaryi*,
methodsi belong to the *methods* list
For i=1..N *smi* is the state machine in the *stateMachines* list defining the behaviour of *Boundaryi*

- **TStaticViewStores**(Sequence(String))

It transforms all the classes with the stereotype ≪store≫ contained in the static view.

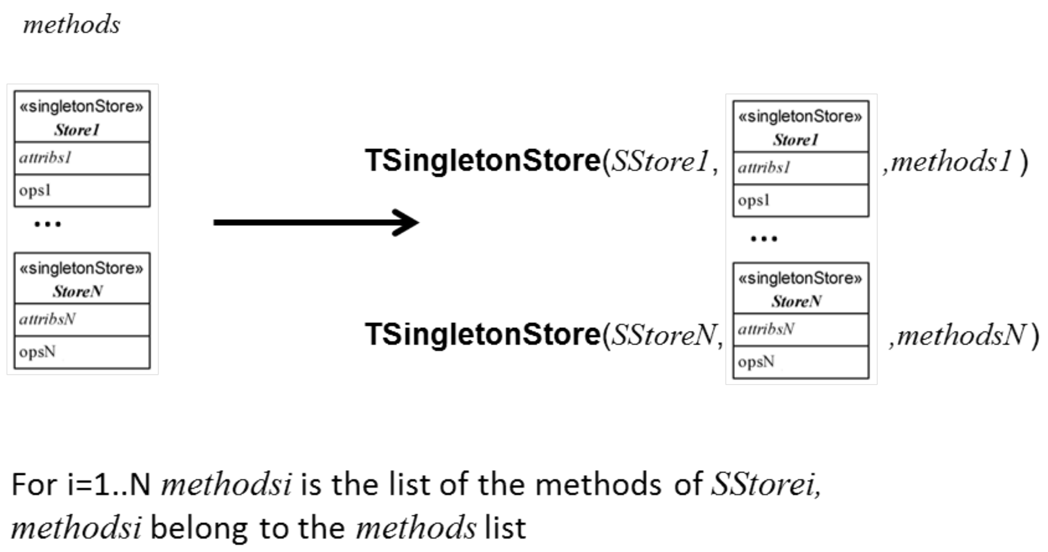
Transformation case



- **TStaticViewSingletonStores**(Sequence(String))

It transforms all the classes with the stereotype ≪singletonStore≫ contained in the static view.

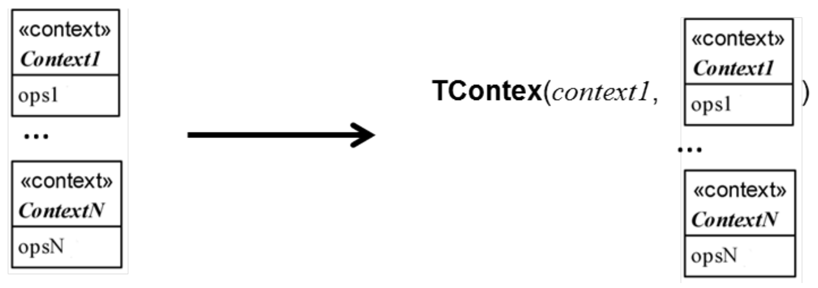
Transformation case



- **TStaticViewContexts**(Sequence(String))

It transforms all the classes with the stereotype ≪context≫ contained in the static view.

Transformation case



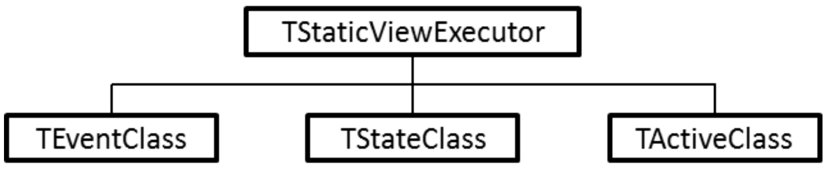
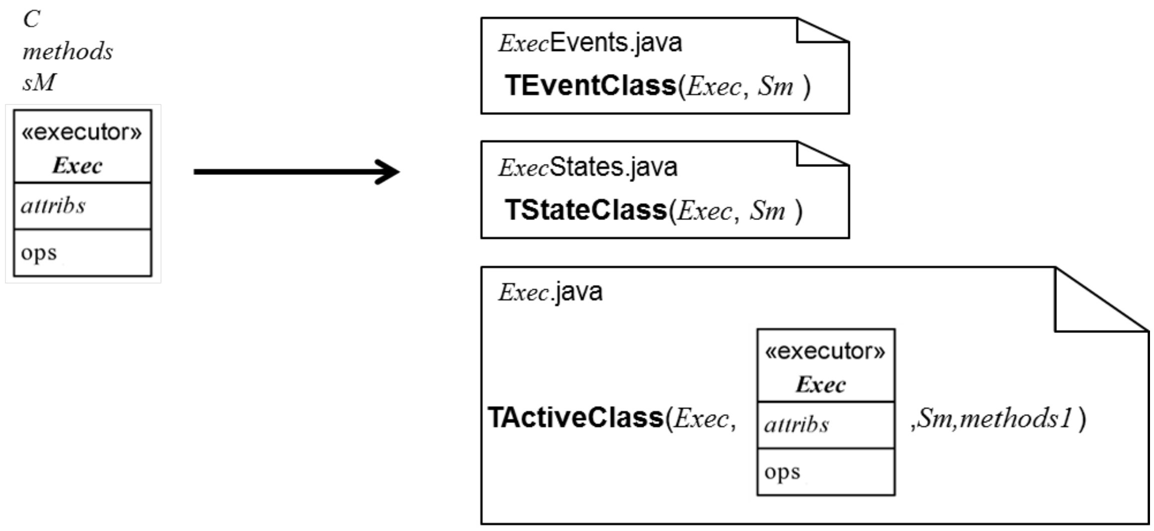


Figure 6.5: U2Java decomposition diagram: TStaticViewExecutor

- **TStaticViewExecutor**(String,ClassDiagram,Sequence(Method),StateMachine)

It transforms an ≪executor≫ entity into the corresponding Java classes and enumerations.

Transformation case



Where: *methods1* is the lists of the methods of *Exec;*
methods1 belong to the *methods* list.

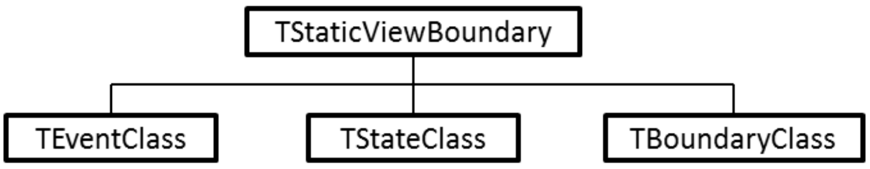
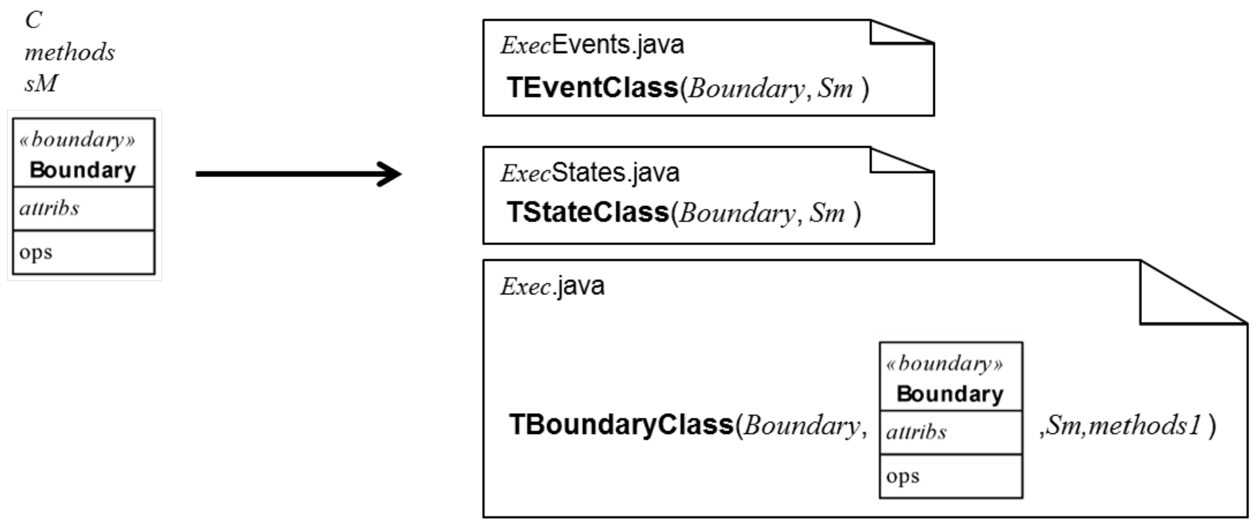


Figure 6.6: U2Java decomposition diagram: TStaticViewBoundary

- **TStaticViewBoundary**(String,ClassDiagram,Sequence(Method),StateMachine)

It transforms an ≪boundary≫ entity into the corresponding Java classes and enumerations.

Transformation case



Where: *methods1* is the lists of the methods of *Boundary;*
methods1 belong to the *methods* list.

6.2.3 Transformation Testing

The test of the U2Java transformation was made before we finalized the definition of our method of testing model transformations as described in Sect. 5.4. We have tested this transformation using an earlier version of our method that was described in our work [106]. Here, we report on that earlier version of our method of testing model transformations and its application to the U2Java transformation.

Kinds of Testing

We consider two main approaches to model transformation testing:

In the former kind of testing we wish to check static properties of the transformation target, thus assess the presence of specific elements in the target; for instance in the case of model to text transformation, it is possible to check the presence in the output of specific strings. This is a kind of white-box testing, indeed we must know the internal structure of the transformation to know which elements search in the target (e.g., if we have to check that some expressions have been correctly transformed in Java, we have to know if the transformation adds extra round parenthesis or if the this are omitted)

In the latter kind of testing we wish to analyse the execution of the transformation target: this can be performed when the target of the transformation is code (so compilable/executable). A test case for the transformation consists of a pair formed by an input model (i.e., a UML Model) and a test case on the target code (e.g., a JUnit test case) that is produced with the help of state-of-the-art techniques in the field of the target of the transformation. A minimal test case just consists in compiling and running the result of the transformation. To try to automatize the generation and the execution of these kind of tests it is possible to define the code test abstractly in the source model and to extend the transformation itself to convert it in a standard test case on the target; for example if the transformation goes from UML models into Java, we can add some parts in the source model and extend the transformation to map them into JUnit tests. This is a kind of black-box testing, at least for the phase of the test cases conception, indeed only the semantics of the transformation and of the target code is considered (e.g., we can ask if the transformation of an expression E will be greater than 0 in the target, just adding in the source model $E > 0$).

Test Models and Oracle Functions

In both approaches we must select input models and oracle functions to be used during the tests. The idea is to build relatively small input models each one used to test a particular kind of possible domain elements instead of large models containing many different elements. For each stereotype in the UML profile used to build the input models there should be at least one test model containing it, and each pattern used in the clauses defining the transformation design should be instantiated in at least one test model.

Checking Static Properties of the Target. In this case, the oracle function analyses the target obtained by a run of the model transformation, asserting the presence of snippets determined analysing the input model. The kind of snippets that must be checked in the target model depends on the patterns specified in the model transformation design, e.g., if in the design there is a clause that states that for each class in the model there must be a class with the same name in the target code, then the oracle function must check the presence of that class.

Analysing the Execution of the Target. In this case we exploit the fact that target produced by the model transformation is compilable and executable. The simplest test is to check if the target code may be compiled. If compilation fails, then we have an indication that the model transformation is incorrect. The model transformation developer can find the erroneous part of the Model transformation with the help of the errors reported by the compiler and the design of the transformation itself.

Another kind of testing is produced inserting in the source model test classes and test operations, that drive the generation of executable test cases in the target code. Tests can be written using specific stereotypes for test classes and test operations, and drawing named associations between the test class and the class under test. Moreover, we can specify post-conditions on test operations indicating what action to take during the build of the generated code, depending if the post-conditions are satisfied or not. More in general, to test semantic properties of the transformation we can write, in the input model, operations whose behaviour is known and test operation to verify that the behaviour is the expected one.

If the execution of the tests generated in the target model fails, the errors can be in the transformation, in the source model or in both. We can consider that in the case of very simple behaviour, qualitatively speaking, the probability of erroneous definition of the model is lower than the probability of erroneous definition of the transformation, thus we consider the transformation incorrect also if in this case we cannot completely exclude errors in the model.

In case of success, we gain in confidence on the correctness of the transformation, except in the case of two errors, one in the source model and the other in the transformation, that compensate each other, but this can be considered a really rare case, also because the errors in the two fields are of a very different nature.

Selection of input models

All the stereotypes that identify the entities composing the applications and their features appear at least in one model in the set of models used for testing the transformation, as well as all the main patterns for the input models appearing in the clauses in the specification of the design of the model transformation itself. Each test model contains mainly elements with a specific class stereotype, so we have:

datatypes test model containing mainly ≪data type≫;

executors test model containing mainly classes with the stereotype ≪executor≫;

boundaries test model containing mainly classes with the stereotype ≪boundary≫;

stores test model containing mainly classes stereotype ≪store≫ and ≪singletonStore≫.

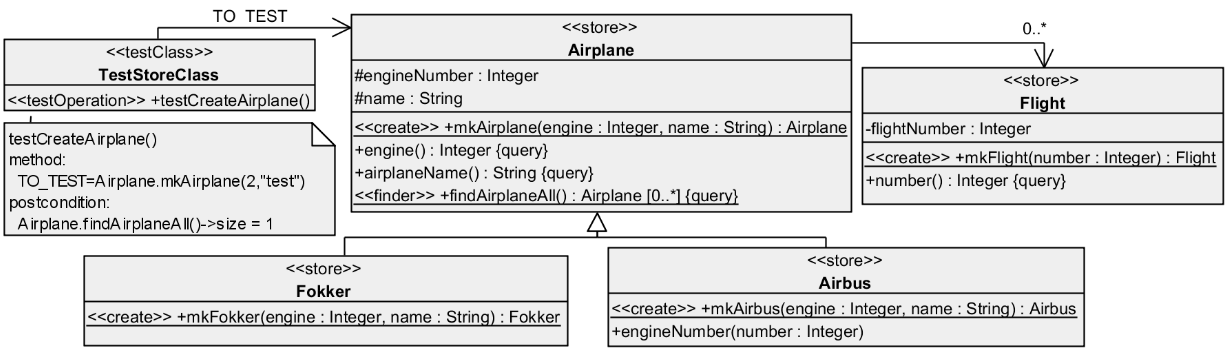

TO TEST
<<testClass>>
TestStoreClass
<<testOperation>> +testCreateAirplane()
testCreateAirplane()
method:
TO_TEST=Airplane.mkAirplane(2,"test")
postcondition:
Airplane.findAirplaneAll()->size = 1
<<store>>
Airplane
#engineNumber : Integer
#name : String
<<create>> +mkAirplane(engine : Integer, name : String) : Airplane
+engine() : Integer {query}
+airplaneName() : String {query}
<<finder>> +findAirplaneAll() : Airplane [0..*] {query}
0..*
<<store>>
Flight
-flightNumber : Integer
<<create>> +mkFlight(number : Integer) : Flight
+number() : Integer {query}
<<store>>
Fokker
<<create>> +mkFokker(engine : Integer, name : String) : Fokker
<<store>>
Airbus
<<create>> +mkAirbus(engine : Integer, name : String) : Airbus
+engineNumber(number : Integer)

Figure 6.7: Fragment of Test Model

Fig. 6.7 shows a fragment of the store test model that contains mainly classes stereotyped with ≪store≫, some parts of this model will be used by the transformation to generate some tests on the target of the transformation itself.

Casses stereotyped by ≪store≫ are transformed in a set of Java classes and configuration files that manage the persistence of objects of these classes in a database. For instance, we can test that the transformation works as expected writing (1) an operation in the input model whose method triggers the creation of one or more row in the database tables and (2) a post-condition to verify that expected data are actually in the database (using for example a query operation of the same stereotyped class). The class *TestStoreClass* stereotyped by ≪testClass≫ contains one operation stereotyped by ≪testOperation≫ (see Fig. 6.7). This operation has a method (shown on the class diagram in the note attached to the operation) that specifies the behaviour of the operation. The operation has one postcondition, specified by an OCL expression (also this one is shown in the note attached to the operation).

During the transformation, if the test generation parameter is enabled, the class stereotyped by ≪testClass≫ is translated into one JUnit test case, and the method associated with each test operation is translated into one JUnit test method. The OCL is compiled in Java language (during the same run of the transformation that generates all the code), and then used to evaluate the post-condition in the JUnit test method. During the build of the project the JUnit test is activated, a fresh instance of an in-memory database is created and configured with scripts automatically derived from the model, and finally the JUnit test is executed. Moreover, it is possible to specify in the model if the build of the generated Maven project must fail in case of test failed, or must produce in any case (test failed or not) the artifacts corresponding to the model.

Regarding the case of checking static properties, if the test generation parameter is enabled, then the transformation can use some elements of the input model (e.g., name of classes, properties of classes etc.) to define parameters for a static analysis tool, and write them in the configuration files parameters of custom check rules) of the Maven project generated by the model transformation under test. These parameters are then used by custom rules of the static analysis tool to verify that the expected snippets are present in the generated code or if there are elements not expected in the generated project. A set of report files are generated during the build of the Maven project, showing the results of applying these rules.

For example, if we want to check the presence of some attributes in a class we can specify the following property:

```
<property name= ''attributes'' value= ''Airplane#name,Airplane#
   engineNumber'' />
```

This property is related to a specific custom rule written for the static analysis tool, that is able to verify the presence of the two attributes, *name* and *engineNumber*, in the class *Airplane*. The rule is then activated during the Maven build and produces files that report the results of its application. This is only a simple example of the possibilities given by the use of a static analysis tools for checking the presence of code snippets in the target project.

6.2.4 Transformation Implementation

In this section, we describe the implementation of the U2Java transformation, using tools and languages selected in Sect. 5.5.

Using these tools and languages, we have implemented:

Well-formedness check It is implemented as an ATL module containing the needed ATL transformations and metamodels.

Refactoring It is implemented as a set of ATL modules (grouped in one ATL eclipse project), one for each refactoring transformation.

TGT Implementation It is implemented as an Acceleo generator which can be launched by means of an eclipse launch configuration or stand alone. Moreover, the generator can be

Tool/ide	Version
Eclipse	Indigo SR2
Acceleo SDK	3.2.1
ATL SDK	3.2.1
Eclipse Modeling Tools	1.4.2

Table 6.2: Tools and Ide versions

```
-- @nsURI UML2=http://www.eclipse.org/uml2/3.0.0/UML
-- @path MM=/wellFormedenessCheck/metamodels/checkConfiguration.ecore
-- @path MM1=/wellFormedenessCheck/metamodels/checkMetaModel.ecore

module MarsWellFormedness;
create OUT: MM1 from IN: UML2, IN1: MM, PROFILE: UML2;

uses stereotypes;
uses marsdesign;
uses configuration;
```

Table 6.3: Well-Formedness ATL Module Header

built using Maven an Tycho[20]

In Table 6.2 are reported the versions of each tools used in the implementation:

Well-Formedness Check Implementation

Here, we describe the eclipse ATL module which implements the well-formedness checks described in Par. Well-Formedness Check. In order to better explain how well-formedness check is implemented using ATL, we show some examples of the ATL code used in this project.

Moreover, the implementation follows the guidelines given in Sect. 5.5.2. It is composed by one ATL module and three libraries.

The ATL module has an header section (showed in Fig. 6.3) referencing the diagnostic and parameter metamodels. Moreover, the construct ***uses*** is used to reference three libraries: *stereotypes*, *marsdesign* and *configuration*. These three libraries contain all the helpers [5] used in the ATL rules contained in this module. The *stereotypes* library contains all the helpers needed to check the presence of one of the stereotypes applied to the source model elements; the *marsdesign* library contains all the helpers needed to evaluate the well-formedness rules over the source model; and the *configuration* library contains all the helpers needed to check the logical value of configuration parameters in the parameter model (conform to the parameter metamodel).

[20]Maven Tycho is a set of Maven plugins and extensions to for building Eclipse plugins and OSGi bundles with Maven. See `eclipse.org/tycho/`

```
rule activeClassWithoutSM {
from
s: UML2!Class,
    p: MM!CheckConfiguration
do {
if(s.activeClassNoSM and
  (p.checkAllFlag or
       p.Syntactic or not
       p.ActiveClassWithoutSM)){
     thisModule.activeClassNoSMElementCreation(s.name);
     }
...
  }
}
```

Table 6.4: Example of Well-Formedness Rule

```
helper context UML2!Class def : activeClassNoSM : Boolean =
  if not self.isActiveClass then false else
  if self.classifierBehavior.oclIsUndefined() then true
  else
   if self.classifierBehavior.oclIsTypeOf(UML2!StateMachine)
     then false
     else true
   endif
  endif
endif;
```

Table 6.5: Example of *marsdesign* library Helper

The rule (*activeClassWithoutSM*) uses one helper (*activeClassNoSM*) and one called-rule (*activeClassNoSMElementCreation*), showed respectively in Fig. 6.5 and in Fig. 6.6. The helper (*activeClassNoSM*), is used to check if in the source model there is a *class* which has not a state machine as behaviour specification. The helper in turn uses another helper (*isActiveClass*) to check if the class has the one of the stereotypes denoting it as an active class (≪executor≫ or ≪boundary≫).

To show how these ATL transformation works in practice, we show a fragment of a source model

```
rule activeClassNoSMElementCreation(clName: String) {
  to
   cont: MM1!ActiveClassNoSM (
    className <- clName,
    description <- 'ERROR: The class ' + clName
            + ' is active but it has no'
                + ' StateMachine'
 + ' as classifierBehavior'
   )
  do {
    cont;
  }
}
```

Table 6.6: Example of Called-Rule

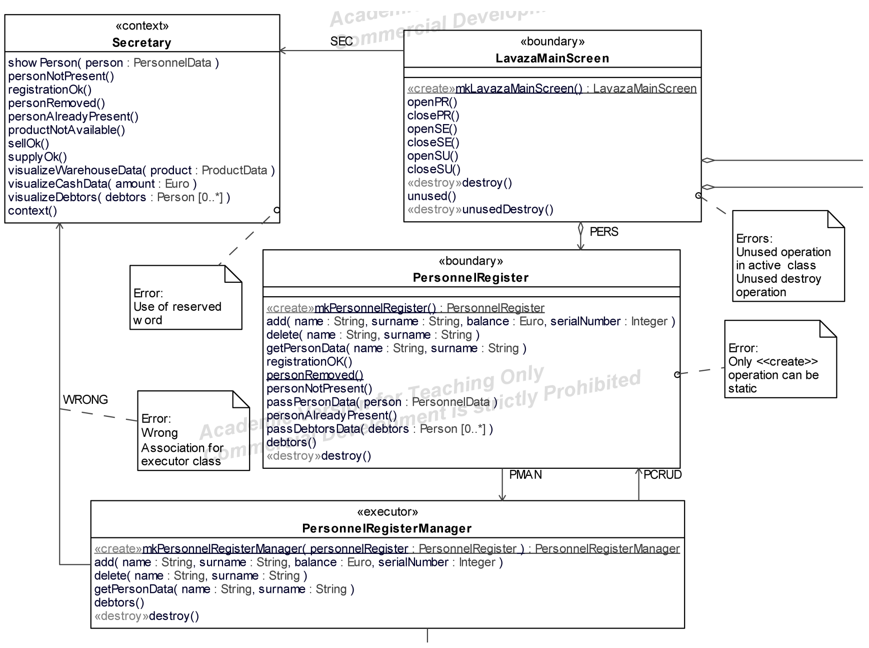
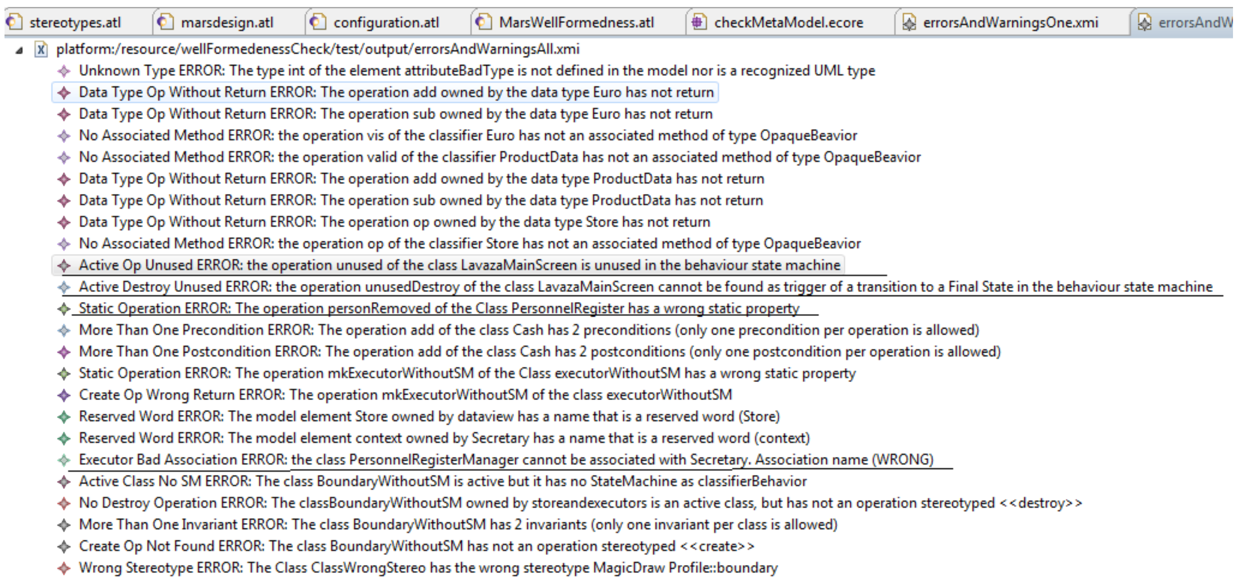

«context»
Secretary
show Person(person : PersonnelData)
personNotPresent()
registrationOk()
personRemoved()
personAlreadyPresent()
productNotAvailable()
sellOk()
supplyOk()
visualizeWarehouseData(product : ProductData)
visualizeCashData(amount : Euro)
visualizeDebtors(debtors : Person [0..*])
context()
SEC
«boundary»
LavazaMainScreen
«create»mkLavazaMainScreen() : LavazaMainScreen
openPR()
closePR()
openSE()
closeSE()
openSU()
closeSU()
«destroy»destroy()
unused()
«destroy»unusedDestroy()
PERS
Errors:
Unused operation in active class
Unused destroy operation
Error:
Use of reserved word
«boundary»
PersonnelRegister
«create»mkPersonnelRegister() : PersonnelRegister
add(name : String, surname : String, balance : Euro, serialNumber : Integer)
delete(name : String, surname : String)
getPersonData(name : String, surname : String)
registrationOK()
personRemoved()
personNotPresent()
passPersonData(person : PersonnelData)
personAlreadyPresent()
passDebtorsData(debtors : Person [0..*])
debtors()
«destroy»destroy()
Error:
Only <<create>> operation can be static
WRONG
Error:
Wrong Association for executor class
PMAN
PCRUD
«executor»
PersonnelRegisterManager
«create»mkPersonnelRegisterManager(personnelRegister : PersonnelRegister) : PersonnelRegisterManager
add(name : String, surname : String, balance : Euro, serialNumber : Integer)
delete(name : String, surname : String)
getPersonData(name : String, surname : String)
debtors()
«destroy»destroy()
Academic Version for Teaching Only
Commercial Development is strictly Prohibited

Figure 6.8: Source Model with Errors



```
platform:/resource/wellFormedenessCheck/test/output/errorsAndWarningsAll.xmi
  Unknown Type ERROR: The type int of the element attributeBadType is not defined in the model nor is a recognized UML type
  Data Type Op Without Return ERROR: The operation add owned by the data type Euro has not return
  Data Type Op Without Return ERROR: The operation sub owned by the data type Euro has not return
  No Associated Method ERROR: the operation vis of the classifier Euro has not an associated method of type OpaqueBeavior
  No Associated Method ERROR: the operation valid of the classifier ProductData has not an associated method of type OpaqueBeavior
  Data Type Op Without Return ERROR: The operation add owned by the data type ProductData has not return
  Data Type Op Without Return ERROR: The operation sub owned by the data type ProductData has not return
  Data Type Op Without Return ERROR: The operation op owned by the data type Store has not return
  No Associated Method ERROR: the operation op of the classifier Store has not an associated method of type OpaqueBeavior
  Active Op Unused ERROR: the operation unused of the class LavazaMainScreen is unused in the behaviour state machine
  Active Destroy Unused ERROR: the operation unusedDestroy of the class LavazaMainScreen cannot be found as trigger of a transition to a Final State in the behaviour state machine
  Static Operation ERROR: The operation personRemoved of the Class PersonnelRegister has a wrong static property
  More Than One Precondition ERROR: The operation add of the class Cash has 2 preconditions (only one precondition per operation is allowed)
  More Than One Postcondition ERROR: The operation add of the class Cash has 2 postconditions (only one postcondition per operation is allowed)
  Static Operation ERROR: The operation mkExecutorWithoutSM of the Class executorWithoutSM has a wrong static property
  Create Op Wrong Return ERROR: The operation mkExecutorWithoutSM of the class executorWithoutSM
  Reserved Word ERROR: The model element Store owned by dataview has a name that is a reserved word (Store)
  Reserved Word ERROR: The model element context owned by Secretary has a name that is a reserved word (context)
  Executor Bad Association ERROR: the class PersonnelRegisterManager cannot be associated with Secretary. Association name (WRONG)
  Active Class No SM ERROR: The class BoundaryWithoutSM is active but it has no StateMachine as classifierBehavior
  No Destroy Operation ERROR: The classBoundaryWithoutSM owned by storeandexecutors is an active class, but has not an operation stereotyped <<destroy>>
  More Than One Invariant ERROR: The class BoundaryWithoutSM has 2 invariants (only one invariant per class is allowed)
  Create Op Not Found ERROR: The class BoundaryWithoutSM has not an operation stereotyped <<create>>
  Wrong Stereotype ERROR: The Class ClassWrongStereo has the wrong stereotype MagicDraw Profile::boundary
```

Figure 6.9: Visualization of the Model Result of the Well-formedness Check

```
-- @nsURI UML2=http://www.eclipse.org/uml2/3.0.0/UML
-- @atlcompiler atl2010

module removeUnnamedAssociations;
create OUT : UML2 refining IN : UML2;

rule deletePropertyOfUnnamedAssociation {
from
    s: UML2!Property,
    s1: UML2!Association
( s1.memberEnd->exists(a |a=s) and  s1.name='')
to
    drop
}

rule deleteUnnamedAssociation {
from
    s1: UML2!Association
( s1.name='')
to
    drop
}
```

Figure 6.10: ATL refactoring transformation

in which are inserted some errors on purpose (see Fig. 6.8) and the model[21] result of the well-formedness check transformation (see Fig. 6.9). Note that in Fig. 6.9 items relative to the errors inserted into the source model fragment are underlined.

Refactoring Implementation

Here, we describe the implementation of the refactorings as designed in Sect. 6.2.2. Given that refactoring operations may be of very different types we do not have a template to follow as in the case of the well-formedness check transformation.

Each reafctoring operation is implemented as an ATL module that can be used independently from the others. In this case we show only one of the ATL transformations, just to highlight some particularity. In Fig. 6.10 is showed the implementation of the refactoring which remove from the source model unnamed associations.

The first thing to note is that in this ATL module is written to be executed in refining mode , thus all the source model elements which remains unchanged between the source and the target model, are implicitly processed (i.e., copied) by the ATL engine. In refining mode the source and the target models are the same. Moreover, we can note that in the ATL module showed in Fig. 6.10 is used the keyword **drop**. The **drop** keyword produce the selected elements for a brief explanation of how declarative ATL rules works) are deleted from the model. It is implemented

[21]To show a model like this in the eclipse environment, we can use the simple Ecore Reflective Editor. It can be activated, simply selecting the model output of the transformation and, by means of the contextual menu, select *open with > simple Ecore reflective editor*.

```
#java source folder for the generated maven projects
java.source.folder = src/main/java
#java resource folder for the generated maven projects
java.resource.folder = src/main/resources
#java test source folder for the generated maven projects
java.test.source.folder = src/test/java
#java test resource folder for the generated maven projects
java.test.resource.folder = src/test/resources

#possible values yes/no
#application context
applicationcontext.database.embedded=yes
applicationcontext.database.access.jpa=yes
applicationcontext.database.transaction=yes
```

Figure 6.11: Properties file

in the ATL2010 version of the compiler. Previous version of the compiler have a different implementation of the refining mode, and do not give support for the **drop** keyword.

Text Generation Transformation implementation

In this section we do not report all the implementations details, but rather we give hints on how Acceleo features has been used. Moreover, we show some examples of the implementation code. The description of these implementation, is obviously dependent from the version of the tools and library used.

In the Text Generation Transformation (TGT) implementation we have used the capability of Acceleo to load properties file[22] to customize the generation. As shown in Fig. 6.11 we use them in several cases, e.g., to specify the root folder in which Java code must be generated, to insert in the generated project the required dependencies or to specify if the data base must be embedded.

Some parts of the source models are expressed using the OCL languages (e.g., invariants, pre and post-conditions, guards), other are expressed using an action language (until now, we have used a Java like action language). All these fragments of the source models must be translated in Java. In the following we describe the techniques used to transform an invariant constraint into Java statements. In order to transform OCL in Java we take advantage of two Acceleo features, the first is the ability of an Acceleo module to have a reference to more than one metamodel and the other is the ability to call a method of a Java object.

The transformation of an invariant constraint written in OCL take place in the following way:

1. a template having as parameter the classifier context of the constraint and the constraint itself, is activated Fig. 6.13. This template is declared in an module which is parametrized

[22]properties files are standard Java properties files which contain key/value pairs, see `docs.oracle.com/javase/tutorial/essential/environment/properties.html`

```
[module compileOCLMature('http://www.eclipse.org/uml2/3.0.0/UML',
                         'http://www.eclipse.org/ocl/1.1.0/UML')]

...
```

Figure 6.12: OCL to Java: Acceleo module definition

```
[template public invariantConstraints(cl:Classifier,constr : Constraint)]
[if (not cl.uml2OclExpression(constr).oclIsUndefined())]
[cl.uml2OclExpression(constr).expr()/]
[else]
  //constraint parse ERROR  ['error'.getError()/]
[/if]
[/template]
```

Figure 6.13: OCL to Java: invariant constraint to Java

by two URI[23], one for the UML metamodel and the other for the OCL metamodel; in this way templates declared inside may have access to element belonging to both metamodels.

2. The template call a Java service (that is method of a Java class) which, using the eclipse OCL library parse the constraint and return a OCL expression object.

3. Another template *expr* (see Fig. 6.13) is activated; this one (and several others who are called by this template) translate the OCL expression in Java. The Java statements generated by this template make use of set of Java classes written to support OCL types and operations.

We want to point out that, since the templates and code were written for the Indigo version of eclipse, the OCL metamodel referenced by templates is the one called *mature eclipse OCL metamodel*[24].

Test Transformations implementation

Test classes (those stereotyped ≪testClass≫) and test operations (those stereotyped ≪testOperation≫) are transformed using Acceleo templates as the other class and operations. In addition to test classes and test operations , the templates generate a JUnit test class (which use the Spring framework) containing a test for each test operation. In each of the generated JUnit test methods, the right test class is instantiated and the right method is called. Since, test on model are modelled with operation postconditions, generated JUnit methods fail if the postcondition throws an exception. In Fig. 6.14 is showed an example (simplified) of a generated JUnit method for the operation *TestMkStoreClassAParams* of the test class *TestStoreClassA*.

[23]en.wikipedia.org/wiki/Uniform_resource_identifier

[24]see the eclipse OCL documentation for the Indigo version help.eclipse.org/indigo/index.jsp

```
  @Test
@Transactional
@Rollback(true)
public void testTestMkStoreClassAParams(){
    TestStoreClassA  classRef = null;
    classRef = new TestStoreClassA();
    try{
     classRef.testMkStoreClassAParams();
    }catch(Exception e){
      fail(e.getMessage()+e.getCause().getMessage());
    }
  }
```

Figure 6.14: Generated test method example

```
  public void visualizeWarehouseData(ProductData product) {
    txtArabicCoffeeSupply.setText(product.getArabicQuantity().toString());
    txtCoffeeSupply.setText(product.getCoffeeQuantity().toString());
    txtTeaSupply.setText(product.getTeaQuantity().toString());
    txtLeamonTeaSupply.setText(product.getLeamonTeaQuantity().toString());
    txtChamomileSupply.setText(product.getChamomileQuantity().toString());
 }
```

Figure 6.15: Example of a method of a **context** entityimplemented in the GUI

Moreover, are also generated all the resources needed for these tests like appropriate Spring [29] application context and JPA [14] persistence configuration. Since the test class and resources are generated in the standard test folder as required by maven, during the build of the generated Java project, tests are executed, and if there is no any type of error, the Java application is built. We do not describe here all the technicalities related with the definition of an appropriate Spring application context, or a JPA configuration.

Manual GUI implementation

Since, *MeDMoT* does not help in the modelling and the implementation of the GUI, to have a complete application we have to manually write the GUI.

After having written the source model, we have used the implementation of U2Java to generate all the code except the GUI, then we have use the hand made GUI to complete the Java desktop application. The project of the complete application is managed by maven, with a dependency with the packaged bytecode of the generated part of the application.

Operations of the **context** entity (modelled as a Java interface) must be implemented in the GUIs (see Fig. 6.15 as an example of the implementation of interface methods).

Moreover, operations of the **boundary** entities must be called from the code which compose the GUI (see Fig. 6.16 as an example of how **boundary** methods are called).

```
   btnShowWarehouseData.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
       lavazaMainScreen.getSUPPL().getWarehouseData();
     }
 });
```

Figure 6.16: Example of a method of a **boundary** entityimplemented in the GUI

6.2.5 The AutoMARS Tool

AutoMARS is a model driven application generator that starting from a UML model representing a detailed design of an application can generate complete (excluding the GUI) Java desktop applications built using up-to-date technologies. All the details of the application can be in the source model, including operation method behaviour and constraints like class invariants, pre-conditions and post-conditions of operations. Also, automated test suite (JUnit test suite) for the generated Java application can be written directly on the source model. The source model is checked against a set of well-formedness constraints, helping to avoid the most common errors. A user may concentrate himself/herself on producing and checking input models, thus increasing the level of abstraction used in the development of software. Moreover, the model transformation that is the core of the tool U2Java, is designed and built following *MeDMoT*. Indeed, AutoMARS is composed by the set of ATL modules implementing the well-formedness check, the refactoring transformations and the Acceleo Generator (until now, we have not implemented a graphic user interface that can activate each ATL transformation/Acceleo generation in the desidered order). Since, this is a prototype tool, not all the well-formedness checks were implemented. Moreover, for the same reason, not all the OCL language expressions can be translated into Java by the Acceleo generator (see Sect. 6.2.4.

AutoMARS were tested on a source model of the design of a Java desktop application. The Java desktop application chosen is a simple application to manage the sale of coffee pods in our department , which we have called *Lavaza*. This shows that our method works also in a not trivial case.

In Fig. 6.17 are showed a snapshots of the Lavaza GUI.

6.3 Conclusion

In this chapter we have presented the application of *MeDMoT* to two case studies: U2OWL and U2Java. The former is a transformation from UML models of ontologies into the corresponding definitions written using the OWL/RDF concrete syntax, the latter is a transformation from UML models of the design of Java desktop applications into Java desktop application built using up-to-date technologies and frameworks.

In the U2OWL case study we have applied all *MeDMoT*, excluding the implementation phase.

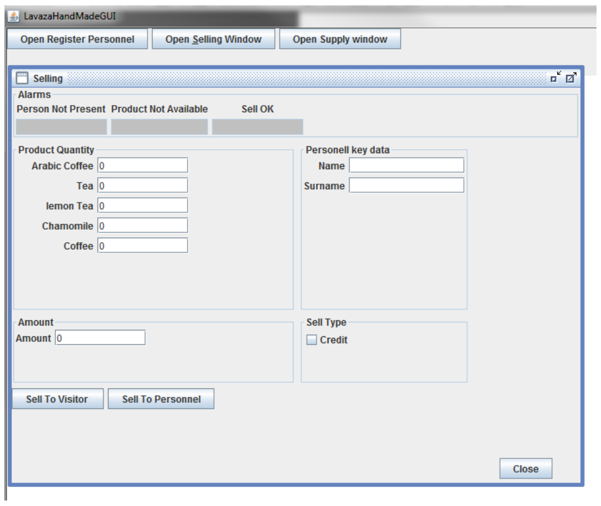


Figure 6.17: Lavaza GUI: Selling

Moreover, we gave the design of the TGT in both the two ways provided by the method : *User Level Design* and *Developer Level Design*. Tests suites were designed but not implemented.

In the U2Java case study we have applied all *MeDMoT*. Due to the size of this case, we gave the design of the TGT only as *User Level Design*. Testing of the TGT were developed and implemented using a testing method developed before the method for testing model transformations as described in Sect. 5.4. Moreover, the implementation of this transformation lead us to the development of a prototypical tool the we have called AutoMARS whose implementation is described in Sect. 6.2.5.

Chapter 7

Conclusion

Model driven techniques aim to rise the level of abstraction in the software development activities and a lot of work has been done in the last fifteen years in study of these techniques. Moreover, modelling and model driven are currently applied by the industry, though not by the majority of software practitioners. Model transformations play a leading role in model driven techniques.

In this thesis we have presented *MeDMoT*, which is a general and effective method to develop model to text transformations. *MeDMoT* can be used to develop model to text transformations not only when the transformation target is code but also when the transformation target is a structured textual artifact (a structured textual artifact is a set of text files, written using one or more concrete syntaxes, disposed in a well-defined structure) with its own semantic. Thus, *MeDMoT* can be used (and has been used) also when the target of the transformation are OWL definitions of ontologies, SQL DDL or a service system implemented using BPEL and web services, text used as input of simulators, documentation and in general when textual artifacts need during the development may be derived by models.

We have pragmatically designed *MeDMoT*, so that it can supports the development of transformations of reasonable size and without requiring at the developer to use a very formal notation. Moreover, we take care of the supporting tools and model transformation languages which are to be used in the implementation phase.

In Chapter 3 we report on our empirical studies on the dissemination and the effectiveness of modelling and model driven techniques during software development and model driven usage. In this chapter, we describe the design, the preparation and the execution of a survey designed to understand the actual diffusion of software modelling and model driven techniques in the Italian industry, and then we describe the findings obtained. These findings confirmed that modelling is widespread and MD* techniques are used by a relevant percentage of who use modelling. Moreover, they use it mainly to capture an high level view of the system and for documentation purposes.

In the same chapter (Chapter 3) we describe an empirical experiment aimed at investigating the effectiveness of model driven development during software maintenance and evolution activities. The results, although preliminary, indicate a relevant shortening of time spent in these activities with no significant impact on correctness.

Since there is the need to apply at model transformation development all the software engineering principles that are applied to the software development, *MeDMoT* covers all the development phases such as: requirements definition, design, implementation and testing. Each of these phases presents problems which have not yet found a standard and shared solution. Thus, for each phase covered by our method we give our contribution.

In the following, we summarise contributions given for each development phases (described in Chapter 4 and in Chapter 5):

Transformation requirements. We have defined a method to capture and describe transformation requirements. As is described in Chapter 5, this method requires the definition of the source, the target of the transformation and a characterization of a relation among source and target elements, expressed in terms of the semantics of these elements. To define the transformation source the method require to use a modeling method following the precise approach, presented in Chapter 4. In the same chapter we present three modeling methods following the precise approach: MRelational a method to model a relational data base using UML, MOntology a method to model ontologies using UML, and MJavaD a method to model the design of desktop applications using the UML.

Model transformation design. We have defined the architecture that model to text transformations developed using *MeDMoT* should have. We give guidelines on how to design each transformation composing the model to text transformation, relaying on the concrete syntax of the source and the target of the transformation. Moreover, we have developed two semi-graphical notations to express the design.

Model transformation implementation. We give guidelines and patterns on how to implement the various parts of the transformation, indicating which tools and transformation languages should be used.

Model transformation testing. Again in Chapter 5, for model to text transformation we propose: (1) a transformation test classification on the basis of the intent with which they are carried out; (2) a new definition of *test adequacy criterion*; (3) a specific definition of *test coverage*; (4) three test adequacy criteria; (5) a method to test the model transformation applying (1),(2),(3) and (4); (5) guidelines on how to automate testing; (6) a way to perform unit tests. Finally, we describe how to test input models.

In Chapter 6 and in Chapter 5 we have validated our method applying it to three case studies of different size. The smallest one is U2SQL, a transformation from UML models of relational data bases to SQL DDL. This case study, that is a very simple one, was used as running example used through the thesis, for this reason it is described in Chapter 5 whereas the other two are described in Chapter 6. The medium size case is U2OWL, a transformation from UML models of ontologies to OWL. Finally, the larger case is U2Java, a transformation from UML models of desktop applications to Java desktop applications. The third case of study leads us to the development of a tool that is basically an implementation of the transformation developed in this case. Our tool, that we call AutoMARS, is a model driven application generator that starting from a UML model representing a detailed design of an application can generate complete (excluding the GUI) Java desktop applications built using up-to-date technologies. All the details of the application can be in the source model, including operation method behaviour and constraints like class invariants, pre-conditions and post-conditions of operations. Also, automated test suite (JUnit test suite) for the generated Java application can be written directly on the source model. The source model is checked against a set of well-formedness constraints, helping to avoid the most common errors. The tool was implemented using ATL and Acceleo and is composed by 61 modules, 403 templates, 73 queries, and 4 Java classes.

As future works, we would like to empirically validate our method to design model to text transformations, to see if this kind of design is more effective than design done at metamodel level (especially in the case of source models written using UML). Moreover, we would like to empirically validate our testing method, by means of mutation analysis and fault injection. We would like also to apply *MeDMoT* to other cases, and to improve the model transformation requirement definition, exploring new ways to characterize the relation between the source and the target of the transformation.

Appendix A

U2OWL: Test suites

A.1 Selection of source models

To apply the three test adequacy criterion described in Sect. 5.4.3 to the MOntology case (see Sect. 4.3) we must define the set *S* of the stereotypes and tagged value comprising the UML profile used for modelling ontologies, thus *S*={ ≪category≫, ≪instances≫ }.

Moreover, we have to define the set *CD* of the UML constructs/diagrams allowed to appear in the source models, considered relevant by the developer, thus *CD*={ class, association, object, link, class diagram, object diagram, generalization/specialization relation }.

Applying criterion 1

Following this criterion we can define eight predicates, one for each element of $M = S \cup CD$.

The following is a list of the source models, that can be generated following this criterion. Note that we give only a brief description of these models.

MOD1 : a class diagram containing only one class stereotyped ≪category≫;

MOD2 : a class diagram containing two classes stereotyped ≪category≫ related by an association;

MOD3 : a class diagram containing two classes stereotyped ≪category≫, one specialization of the other;

MOD4 : a class diagram containing only one class stereotyped ≪instances≫;

MOD5 : a class diagram containing only one class stereotyped ≪category≫ and an object diagram with one object instance of the class contained in the class diagram;

MOD6 : a class diagram containing two classes stereotyped ≪category≫ related by an association and an object diagram with two object instances of the classes contained in the class diagram and connected with a link instance of the association;

MOD7 : a class diagram containing two classes stereotyped ≪category≫, one specialization of the other and an object diagram containing two object instances of the class contained in the class diagram;

MOD8 : a class diagram containing: two classes stereotyped ≪category≫ related by an association, another two classes stereotyped ≪category≫ one specialization of the other. An object diagram with all the objects instances of the classes contained in the class diagram and a link instance of the association.

Applying criterion 2

In this case we must define what features of the elements of the set M are relevant. In the following tables are showed the elements of M chosen, for each element are defined the features to take into consideration and their relevant variation.

Element	features	relevant variation
class	class attributes	all primitive types
association	starting and ending multiplicity	1-*; *-*; *-1
class	class attributes	at least three attributes
object	slots	all primitive types

Table A.1: Feature sets

The following is a list of the source models, that can be generated following this criterion:

MOD1 : a class diagram containing only one class stereotyped ≪category≫. The class has a number of attributes equal to the number of primitive types.

MOD2 : a class diagram containing six classes stereotyped ≪category≫ and three associations that relate the classes two by two. Associations are characterized by the following starting and ending multiplicity: 1-*,*-* and *-1;

MOD3 : a class diagram containing only one class stereotyped ≪instances≫ containing at least three attributes;

MOD4 : a class diagram containing only one class stereotyped ≪category≫. The class has a number of attributes equal to the number of primitive types. An object diagram with one object instance of the class contained in the class diagram, with all the slots containing a value.

Applying criterion 3

The source models are instances of the templates used in the design of the text generation transformation (see Sect. 6.1.2).

Bibliography

[1] Acceleo. http://www.eclipse.org/acceleo/. Accessed: 20/02/2014.

[2] Apache maven project. http://maven.apache.org/. Accessed: 20/02/2014.

[3] Atl. http://www.eclipse.org/atl/. Accessed: 20/02/2014.

[4] Atl called rules. http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language#Called_Rules. Accessed: 20/02/2014.

[5] Atl helpers. http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language#ATL_Helpers. Accessed: 20/02/2014.

[6] Atl library. http://wiki.eclipse.org/ATL/User_Guide_-_Overview_of_the_Atlas_Transformation_Language#ATL_Library. Accessed: 20/02/2014.

[7] Atl matched rules. http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language#Matched_Rules. Accessed: 20/02/2014.

[8] Atl refining mode. http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language#ATL_Refining_Mode. Accessed: 20/02/2014.

[9] Eclipse ecore tools. http://wiki.eclipse.org/index.php/Ecore_Tools. Accessed: 20/02/2014.

[10] Eclipse modeling framework. http:http://www.eclipse.org/modeling/emf/. Accessed: 20/02/2014.

[11] Eclipse modeling project. http://www.eclipse.org/modeling/. Accessed: 20/02/2014.

[12] Eclipse platform. http://www.eclipse.org/. Accessed: 20/02/2014.

[13] Hibernate relational persistance for java and .net. http://www.hibernate.org/. Accessed: 20/02/2014.

[14] Java persistent api. http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html. Accessed: 20/02/2014.

[15] Jet2. http://www.eclipse.org/modeling/m2t/?project=jet. Accessed: 20/02/2014.

[16] Kermeta. http://www.kermeta.org/. Accessed: 16/02/2014.

[17] Meta object facility (mof) 2.0 query/view/transformation, v1.1. http://www.omg.org/spec/QVT/1.1/. Accessed: 16/02/2014.

[18] Ocl. http://www.eclipse.org/modeling/mdt/?project=ocl. Accessed: 20/02/2014.

[19] Omg common warehouse metamodel. http://www.omg.org/spec/CWM/. Accessed: 20/02/2014.

[20] Omg metaobject facility. http://www.omg.org/mof/. Accessed: 20/02/2014.

[21] Omg mof 2 xmi mapping. http://www.omg.org/spec/XMI/. Accessed: 20/02/2014.

[22] Omg mof model to text transformation language (mofm2t). http://www.omg.org/spec/MOFM2T/1.0/. Accessed: 20/02/2014.

[23] Omg mof query/view/transform qvt. http://www.omg.org/technology/documents/modeling_spec_catalog.htm#QVTL. Accessed: 20/02/2014.

[24] Omg object constraint language. http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL. Accessed: 20/02/2014.

[25] Omg unified modeling language uml 2.x. http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML. Accessed: 20/02/2014.

[26] Papyrous. http://www.eclipse.org/papyrus/. Accessed: 20/02/2014.

[27] Qvtd. http://www.eclipse.org/mmt/?project=qvtd. Accessed: 20/02/2014.

[28] Qvto. http://www.eclipse.org/mmt/?project=qvto. Accessed: 20/02/2014.

[29] Spring application development framework. http://www.springsource.org/. Accessed: 20/02/2014.

[30] sql-92 bnf. http://savage.net.au/SQL/sql-92.bnf.html. Accessed: 20/02/2014.

[31] Uml profile for enterpise application integration. `http://www.omg.org/spec/EAI/`. Accessed: 20/02/2014.

[32] Uml profile for marte: Modeling and analysis of real-time embedded systems. `http://www.omg.org/spec/MARTE/`. Accessed: 20/02/2014.

[33] Uml profile for modelling qos and fault tolerance characteristics and mechanisms. `http://www.omg.org/spec/QFTP/`. Accessed: 20/02/2014.

[34] Uml profile for schedulability, performance and time. `http://www.omg.org/spec/SPTP/`. Accessed: 20/02/2014.

[35] Uml profile for system on a chip. `http://www.omg.org/spec/SoCP/`. Accessed: 20/02/2014.

[36] Uml testing profile. `http://www.omg.org/spec/UTP/`. Accessed: 1/07/2013.

[37] Uml2. `http://www.eclipse.org/modeling/mdt/?project=uml2`. Accessed: 20/02/2014.

[38] Viatra. `http://www.eclipse.org/viatra2/`. Accessed: 16/02/2014.

[39] Xpand. `http://www.eclipse.org/modeling/m2t/?project=xpand`. Accessed: 20/02/2014.

[40] Software engineering - product quality, ISO/IEC 9126-1. Technical report, International Organization for Standardization, 2001.

[41] Roberto Acerbis, Aldo Bongio, Marco Brambilla, and Stefano Butti. Webratio 5: An eclipse-based case tool for engineering web applications. In Luciano Baresi, Piero Fraternali, and Geert-Jan Houben, editors, *ICWE*, volume 4607 of *Lecture Notes in Computer Science*, pages 501–505. Springer, 2007.

[42] David Ameller. Considering Non-Functional Requirements in Model-Driven Engineering. Master's thesis, Llenguatges i Sistemes Informàtics (LSI), June 2009.

[43] Moussa Amrani, Jürgen Dingel, Leen Lambers, Levi Lúcio, Rick Salay, Gehan Selim, Eugene Syriani, and Manuel Wimmer. Towards a model transformation intent catalog. In *Proc. of AMT 2012*, pages 3–8. ACM, 2012.

[44] Étienne André, Christine Choppy, and Gianna Reggio. Activity diagrams patterns for modeling business processes. In *Software Engineering Research, Management and Applications*, pages 197–213. Springer, 2014.

[45] Egidio Astesiano and Gianna Reggio. Tight structuring for precise UML-based requirement specifications. In Martin Wirsing, Alexander Knapp, and Simonetta Balsamo, editors, *RISSEF*, volume 2941 of *Lecture Notes in Computer Science*, pages 16–34. Springer, 2002.

[46] Egidio Astesiano and Gianna Reggio. Towards a well-founded UML-based development method. In *SEFM*, page 102. IEEE Computer Society, 2003.

[47] Egidio Astesiano, Gianna Reggio, and Maura Cerioli. From formal techniques to well-founded software development methods. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *Lecture Notes in Computer Science*, pages 132–150. Springer, 2002.

[48] C. Atkinson and T. Kuhne. Model-driven development: a metamodeling foundation. *Software, IEEE*, 20(5):36–41, 2003.

[49] V. Basili, G. Caldiera, and D. H. Rombach. *The Goal Question Metric Paradigm, Encyclopedia of Software Engineering*. John Wiley and Sons, 1994.

[50] Benoit Baudry, Trung Dinh-trong, Jean marie Mottu, Devon Simmonds, Robert France, Sudipto Ghosh, Franck Fleurey, and Yves Le Traon. Traon. model transformation testing challenges. In *In Proceedings of IMDT workshop in conjunction with ECMDA06*, 2006.

[51] Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert France, Yves Le Traon, and Jean-Marie Mottu. Barriers to systematic model transformation testing. *Commun. ACM*, 53(6):139–143, 2010.

[52] Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert B. France, Yves Le Traon, and Jean-Marie Mottu. Barriers to systematic model transformation testing. *Commun. ACM*, 53(6):139–143, 2010.

[53] J Bezivin and F Jouault. Using atl for checking models. *Electronic Notes in Theoretical Computer Science*, 152(0):69–81, 2006.

[54] Jean Bézivin. In Search of a Basic Principle for Model Driven Engineering. *UPGRADE – The European Journal for the Informatics Professional*, 5(2):21–24, 2004.

[55] Jean Bézivin. Model driven engineering: an emerging technical space. In *Proceedings of the 2005 international conference on Generative and Transformational Techniques in Software Engineering*, GTTSE'05, pages 36–64, Berlin, Heidelberg, 2006. Springer-Verlag.

[56] Jean Bézivin and Frédéric Jouault. Using ATL for checking models. *Electr. Notes Theor. Comput. Sci*, 152:69–81, 2006.

[57] Matthias Biehl. Literature Study on Model Transformations. Technical Report ISRN/KTH/MMK/R-10/07-SE, Royal Institute of Technology, July 2010.

[58] Alan W. Brown. Model driven architecture: Principles and practice. *Software and Systems Modeling*, 3(4):314–327, 2004.

[59] L. Burgueño and M. Wimmer. Testing M2T/T2M transformations. In *ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems 2013 (MoDELS 2013)*, Miami, USA, September-October 2013. to appear.

[60] M. Ciolkowski, O. Laitenberger, and S. Biffl. Software reviews: The state of the practice. *IEEE Software*, 20(6):46–51, Nov/Dec 2003.

[61] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, July 2006.

[62] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[63] Hans-Erik Eriksson, Magnus Penker, and David Fado. *UML 2 Toolkit*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[64] Franck Fleurey, Zoé Drey, Didier Vojtisek, Cyril Faucher, and Vincent Mahé. Kermeta language, reference manual. *Internet: http://www. kermeta. org/docs/KerMeta-Manual. pdf. IRISA*, 2006.

[65] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003.

[66] L. Fuentes-Fernández and A. Vallecillo-Moreno. An Introduction to UML Profiles. *UPGRADE, European Journal for the Informatics Professional*, 5(2):5–13, April 2004.

[67] Pau Giner and Vicente Pelechano. Test-driven development of model transformations. In Andy Schürr and Bran Selic, editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 2009.

[68] Object M. Group. OMG Unified Modeling Language (OMG UML), Superstructure, V2.4.1. Technical report, August 2011.

[69] Object M. Group. OMG Object Constraint Language (OCL) V2.3.1. Technical report, January 2012.

[70] Esther Guerra, Juan de Lara, Dimitrios Kolovos, Richard Paige, and Osmar dos Santos. Engineering model transformations with *trans*ML. pages 1–23, 2011.

[71] Esther Guerra, Juan de Lara, Dimitrios S. Kolovos, Richard F. Paige, and Osmar Marchi dos Santos. trans ML: A family of languages to model model transformations. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *MoDELS (1)*, volume 6394 of *Lecture Notes in Computer Science*, pages 106–120. Springer, 2010.

[72] V. Gurov, M. Mazin, A. Narvsky, and A. Shalyto. Tools for support of automata-based programming. *Programming and Computer Software*, 33:343–355, 2007.

[73] B. Hailpern and P. Tarr. Model-driven development: the good, the bad, and the ugly. *IBM Syst. J.*, 45(3):451–461, July 2006.

[74] Philipp Huber. The Model Transformation Language Jungle - An Evaluation and Extension of Existing Approaches. Master's thesis, Technische Universität Wien, May 2008.

[75] Frdric Jouault and Ivan Kurtev. Transforming models with atl. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer Berlin Heidelberg, 2006.

[76] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(12):31 – 39, 2008. Special Issue on Second issue of experimental software and toolkits (EST).

[77] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. Atl: a qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 719–720, New York, NY, USA, 2006. ACM.

[78] Frédéric Jouault and Ivan Kurtev. On the architectural alignment of atl and qvt. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1188–1195, New York, NY, USA, 2006. ACM.

[79] Stuart Kent. Model driven engineering. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 286–298, London, UK, UK, 2002. Springer-Verlag.

[80] B. Kitchenham and S.L. Pfleeger. Personal opinion surveys. In Forrest Shull and Singer, editors, *Guide to Advanced Empirical Software Engineering*, pages 63–92. Springer London, 2008.

[81] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture—Practice and Promise*. Addison-Wesley, 2003.

[82] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[83] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. Technological spaces: An initial appraisal. In *CoopIS, DOA&#039;2002 Federated Conferences, Industrial track*, 2002.

[84] T. C. Lethbridge. A survey of the relevance of computer science and software engineering education. In *Proceedings of the 11th Conference on Software Engineering Education and Training*, pages 0056–, Washington, DC, USA, 1998. IEEE Computer Society.

[85] Ashley McNeile. Mda: The vision with the hole? *Program*, 2003.

[86] MDA. Model driven architecture - a technical perspective. Technical Report ab/2001-02-01, OMG, 2001. Architecture Board MDA Drafting Team Review Draft.

[87] S.J. Mellor, A.N. Clark, and T. Futagami. Model-driven development - guest editor's introduction. *Software, IEEE*, 20(5):14–18, Sept 2003.

[88] T. Mens and T. Tourwe. A survey of software refactoring. *Software Engineering, IEEE Transactions on*, 30(2):126 – 139, feb 2004.

[89] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, March 2006.

[90] P. Mohagheghi and V. Dehlen. Where is the proof? - a review of experiences from applying MDE in industry. In *Proc. of the 4th European conference on Model Driven Architecture: Foundations and Applications*, ECMDA-FA '08, pages 432–443, Berlin, Heidelberg, 2008. Springer-Verlag.

[91] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. Model transformation testing: oracle issue. In *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, ICSTW '08, pages 105–112, Washington, DC, USA, 2008. IEEE Computer Society.

[92] Tim OBrien, John Casey, Brian Fox, Jason Van Zyl, Manfred Moser, Eric Redmond, and Larry Shatzer. Maven: The complete reference. *Online Book*, 2010.

[93] Gianna Reggio, Maurizio Leotta, Filippo Ricca, and Egidio Astesiano. Business process modelling: Five styles and a method to choose the most suitable one. In *Proceedings of the Second Edition of the International Workshop on Experiences and Empirical Studies in Software Modelling*, EESSMod '12, pages 8:1–8:6, New York, NY, USA, 2012. ACM.

[94] Gianna Reggio, Filippo Ricca, Giuseppe Scanniello, FrancescoDi Cerbo, and Gabriella Dodero. On the comprehension of workflows modeled with a precise style: results from a family of controlled experiments. *Software & Systems Modeling*, pages 1–24, 2013.

[95] F. Ricca, M. Leotta, G. Reggio, A. Tiso, G. Guerrini, and M. Torchiano. Using unimod for maintenance tasks: An experimental assessment in the context of model driven development. In *Modeling in Software Engineering (MISE), 2012 ICSE Workshop on*, pages 77 –83, june 2012.

[96] D.C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, Feb 2006.

[97] Andy Schrr. Specification of graph translators with triple graph grammars. In ErnstW. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer Berlin Heidelberg, 1995.

[98] E. Seidewitz. What models mean. *Software, IEEE*, 20(5):26–32, 2003.

[99] B. Selic. Model-driven development: its essence and opportunities. In *Object and Component-Oriented Real-Time Distributed Computing, 2006. ISORC 2006. Ninth IEEE International Symposium on*, pages 7 pp.–, 2006.

[100] S. Sendall and W. Kozaczynski. Model transformation: the heart and soul of model-driven software development. *Software, IEEE*, 20(5):42–45, 2003.

[101] A. A. Shalyto and N. I. Tukkel. SWITCH Technology: An Automated Approach to Developing Software for Reactive Systems. *Programming and Computer Software*, 27:260–276, 2001.

[102] Y Singh and M Sood. *Model Driven Architecture: A Perspective*, pages 1644–1652. Number No. ab/2001-02-04. IEEE, 2009.

[103] Richard Soley. Model driven architecture. Technical report, Object Management Group (OMG), 2000.

[104] Gabriel Tamura, Anthony Cleve, et al. A comparison of taxonomies for model transformation languages. *Paradigma*, 4(1):1–14, 2010.

[105] Massimo Tisi, Salvador Martínez, Frédéric Jouault, and Jordi Cabot. Refining Models with Rule-based Model Transformations. Rapport de recherche RR-7582, INRIA, March 2011.

[106] Alessandro Tiso, Gianna Reggio, and Maurizio Leotta. Early experiences on model transformation testing. In *Proceedings of the First Workshop on the Analysis of Model Transformations*, AMT '12, pages 15–20, New York, NY, USA, 2012. ACM.

[107] Alessandro Tiso, Gianna Reggio, and Maurizio Leotta. A method for testing model to text transformations. In *AMT 2013 - 2nd Workshop on the Analysis of Model Transformations*. CEUR Workshop Proceedings, 2013.

[108] Federico Tomassetti, Marco Torchiano, Alessandro Tiso, Filippo Ricca, and Gianna Reggio. Maturity of software modelling and model driven engineering: A survey in the italian industry. In *Evaluation Assessment in Software Engineering (EASE 2012), 16th International Conference on*, pages 91 –100, may 2012.

[109] Marco Torchiano, Federico Tomassetti, Filippo Ricca, Alessandro Tiso, and Gianna Reggio. Preliminary findings from a survey on the md state of the practice. In *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement*, ESEM '11, pages 372–375, Washington, DC, USA, 2011. IEEE Computer Society.

[110] Marco Torchiano, Federico Tomassetti, Filippo Ricca, Alessandro Tiso, and Gianna Reggio. Benefits from modelling and mdd adoption: expectations and achievements. In *Proceedings of the Second Edition of the International Workshop on Experiences and Empirical Studies in Software Modelling*, EESSMod '12, pages 1:1–1:6, New York, NY, USA, 2012. ACM.

[111] Frank Truyen. The Fast Guide to Model Driven Architecture - The basics of Model Driven Architecture, January 2006.

[112] Dmitry Tsarkov and Ian Horrocks. Fact++ description logic reasoner: system description. In *Proc. of IJCAR 2006*, pages 292–297. Springer, 2006.

[113] M D A Guide Version, Alan Kennedy, Kennedy Carter, and William Frank X-change Technologies. Mda guide version 1.0.1. *Object Management Group*, 234(June):51, 2003.

[114] Markus Völter. MD* best practices. *Journal of Object Technology*, 8(6):79–102, 2009.

[115] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers, 2000.

[116] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.