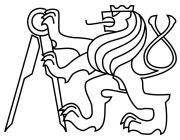
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	Model-Driven Engineering Approach for OntoUML
Student:	Bc. Dan Homola
Supervisor:	Ing. Robert Pergl, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2016/17

Instructions

OntoUML is a high-level ontologically well-founded method for conceptual modelling. Its application in software engineering needs to build on model-driven approach to elaborate several necessary layers between the models and the implementation. The goal of this thesis is to design and implement a semi-automated transition from OntoUML into object-oriented (OO) code. This transition should take into account the existing variety of programming languages. Current existing approaches should be examined and evaluated and possibly used. The considered OntoUML part should cover most of the software engineering needs (refer to the BI-OMO course). The resulting solution should consist of a general layer applicable for most OO languages. Next, develop a solution for a specific arbitrary OO language syntax.

References

Will be provided by the supervisor.

L.S.

Ing. Michal Valenta, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague December 6, 2015

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Master's thesis

Model-Driven Engineering Approach for OntoUML

Bc. Dan Homola

Supervisor: Ing. Robert Pergl, Ph.D.

1st May 2016

Acknowledgements

I would like to thank my supervisor, Ing. Robert Pergl, Ph.D., for his patient and plentiful support and my friend, Ing. Tomáš Doležal, for invaluable advice with some C# details. Many thanks also belong to my family and friends for emotional support and motivation.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 1st May 2016 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Homola, Dan. *Model-Driven Engineering Approach for OntoUML.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstrakt

V této práci je představeno OntoUML a možnosti jeho využití v Model-Driven Engineering přístupu. Je popsán sled tranformací, které vedou od konceptuálního OntoUML modelu ke zdrojovému kódu objektově orientovaného jazyka a je pro něj vytvořena aplikační infrastruktura. Její vhodnost je následně demonstrována implementací transformace OntoUML modelu ze zvolené vstupní formy do zdrojových kódů v jazyce C#.

Klíčová slova Model-Driven Engineering, OntoUML, transformace modelů, C#, generování kódu

Abstract

In this thesis, OntoUML and possibilities of its usage in Model-Driven Engineering are introduced. A set of transformations from OntoUML conceptual model into object-oriented source code is described and an application framework is created for it. The viability of it is then illustrated by implementing the transformation from OntoUML in a concrete input form into C# source code.

Keywords Model-Driven Engineering, OntoUML, model transformation, C#, code generation

Contents

List of Figures

List of Tables

List of Listings

Introduction

OntoUML is a very useful language for conceptual modelling that provides a way to create semantically precise models. However, to use it with Model-Driven Engineering approach is very impractical and inefficient because there are very few tools that even support it and fewer that are capable to generate source code from it (for example OWL files). To my knowledge there is no tool capable of directly generating object-oriented source code from OntoUML models.

Goals and methodology

This thesis aims to create a semi-automated tool that enables transformation from OntoUML conceptual models into target object-oriented language source code files. This should facilitate the usage of OntoUML in Model-Driven Engineering. The main requirements of the system are:

1. Make as little assumptions about the input and output forms as possible

2. Make the system extensible, i.e. easy to add support for another input or output language

Very little assumptions are made about the input form — the only one being that it can be transformed into a valid OntoUML model. Same goes for the output form — it has to be an object-oriented language.

The system is also to be designed in a way that facilitates adding support for other input and output forms than those implemented in this thesis. This is achieved by abstracting the main transformation from OntoUML into Object Model representation (OntoObjectModel) from the input and output forms and separating the parsing of input file and generating of the output file or files. This allows for easy way to add support for another input and/or output form while being able to utilize already existing ones.

Methodology

First, existing solutions were reviewed. Then, intermediate model forms were created and the transformations between them defined. Finally, application that performs these transformations was written to illustrate the viability of the approach.

The application was written in TypeScript 1.8[1] for the Node.js platform 5.10[2] in Visual Studio Code[3] and its C# generator generates C# 6.0 code, so it needs Visual Studio 2015[4] or newer to be compiled (see section 4.5). The utility C# library was written as a NuGet package[5] using Visual Studio 2015 for the .NET platform 4.5.2[6], the tests for it were written using the xUnit.net test framework[7]. The illustrative models were created using OLED editor[8] and diagrams were drawn in UMLet[9] using either OntoUML notation or Yourdon Data Flow Diagram notation[10]. The text was written and typeset in TeXstudio[11].

Thesis structure

In Part I, theoretical basis is established and existing approaches are described and compared. Part II is dedicated to the description of the solution chosen for this thesis – all the intermediate forms and their transformations are specified and also the architecture of the implementation and some of its details are discussed. Part III discusses the results of the provided solution along with its benefits and negatives and potential ways to expand upon them.

Part I

Review

Chapter 1

Model-Driven Engineering

Model-Driven Engineering (MDE) is an approach to software development that is based around domain models. These models are created on a conceptual level, therefore they are independent on the final implementation. After they are validated, they can be transformed into more concrete levels (see section 1.1.3). This very early validation aims to find errors in the solution's design as soon as possible hence reducing costs (the sooner in the development an error is found the cheaper it is to fix it [12]).

MDE is an abstract methodology so initiatives were created based on it that try to formalize MDE's methods and principles. One of the better known, that we will talk about in this thesis, is Model-Driven Architecture.

1.1 Model-Driven Architecture

Model-Driven Architecture (MDA) is an implementation of MDE principles created by OMG. It is built on other OMG standards such as UML, XMI and others. The main goal is to provide 'an approach for deriving value from models and architecture' ([13]).

In other words, this methodology puts accent on formalized models of a system as the ultimate source for all the software development life-cycle activities including design, implementation, deployment and documentation. Also it exploits various types of transformations of those models. Some of the proclaimed benefits of using MDA are[14]:

- Portability and technology obsolescence – the core of the system is described in a way that is independent on the target platform, hence any changes of the platform (be it because of business needs or technological reasons) is much easier
- Productivity – using automated transformations removes the need of some programming and thus enables the developers to put their effort to more crucial parts of the system

- Quality – artifices produced from formally defined models are reliable and contribute to improved quality

In the following sections we will discuss the key notions of MDA relevant to this thesis.

1.1.1 System and Environment

As MDA was created primarily for software development, one might assume that system in this context means computer or software system. However this term is defined much more broadly as 'a collection of parts and relationships organized to accomplish some purpose' ([15]). With this definition, system can contain not only computers and software but also people, companies and even groups of companies.

Environment is everything external to the system that interacts with it. Interesting thought here is that a system can play a role of an environment for another system allowing us to apply the same principles recursively where appropriate.

1.1.2 Model and Metamodel

Model is a structured representation of a system or its part, that is created to address certain aspects of the modelled system (e.g. UML model, DEMO model, etc.). Metamodel is a model of another model's language and syntax that establishes rules every model in that language must conform to.

1.1.3 Viewpoint and View

Viewpoint is a level of abstraction defined by a set of rules and criteria for model creation allowing us to focus only on certain aspects of the system ignoring others. It is realized by a view i.e. one or more models. MDA defines three basic viewpoints (in decreasing level of abstraction):

1. Computation Independent
2. Platform Independent
3. Platform Specific

The most abstract viewpoint is Computation Independent. Models in views conforming to this viewpoint[1] are created in such a way that they do not imply their implementation has to use any certain way (mostly computer systems). These are typically conceptual models and are focused on requirements, ignoring processing etc.

[1]For the sake of brevity we will refer to models in views conforming to a certain viewpoint as viewpoint models from now on.

More concrete viewpoint is called Platform Independent. Its models are constrained by the general way the system is to be implemented (a computer system) but is agnostic to the concrete platforms used by the implementation and describes the operation side of things.

Platform Specific models extend the Platform Independent ones with details regarding the set of technologies selected for the implementation (type of database, programming platform, communication protocols etc.).

Chapter 2

OntoUML

OntoUML is a modelling language based on Unified Foundation Ontology (UFO) [16] built as an UML Profile. It makes use of cognitive psychology — i.e. 'the study of higher mental processes such as attention, language use, memory, perception, problem solving, and thinking' ([17]) — and modal logic. It was first introduced in G. Guizzardi's PhD. thesis [18] and today is further developed by NEMO research group (Guizzardi is a member) [19].

2.1 Entity Types

OntoUML defines several types of entities that can be used in diagrams which can be seen in grey in fig. 2.1. They are divided in two main groups – Sortals and Nonsortals (Mixins). The difference is that Sortal types provide ontological identity to their entities whereas Nonsortals do not. As opposed to UML, in

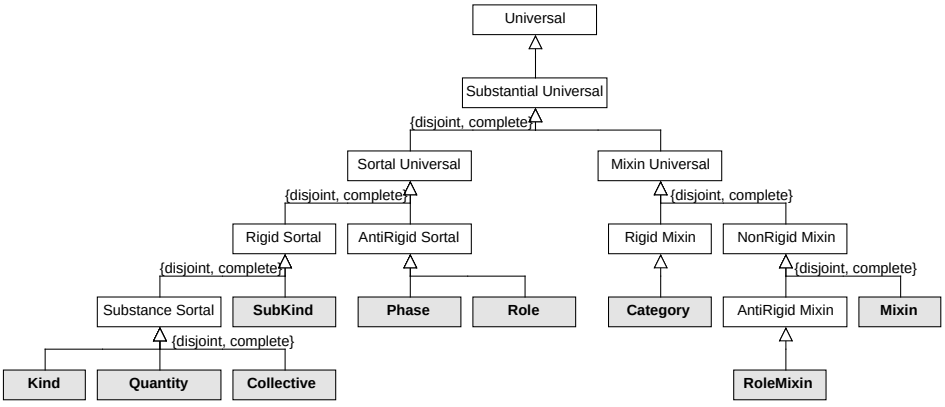


Figure 2.1: OntoUML type hierarchy (adapted from [20])

OntoUML an object can be instance of multiple types (classes) simultaneously but they must form a generalization tree with the root providing the object's identity. This implies that generalization and generalization sets are used more often in OntoUML. There is also a slight difference in the notation of the completeness and disjointness of members of a generalization set – they are incomplete and overlapping by default, any differences must be explicitly marked by `complete` and `disjoint` attributes respectively. The semantics are the same as in UML, though.

Another type of division is according to the type's rigidity. If an object is an instance of a rigid type, this fact cannot change over time. On the other hand, object may become (or cease to be) an instance of an anti-rigid or non-rigid type over time. In the following subsections we will briefly introduce all of OntoUML's types.

2.1.1 Kind and Subkind

Kind and Subkind are rigid sortal types. Kind is the most general type that provides identity to its instances. Subkind represents a specialization of a Kind and must have exactly one Kind as its ancestor to inherit its identity from. Both often serve as ancestors (and therefore sources of identity) to other types.

2.1.2 Role and Phase

Role and Phase are antirigid sortal types. They cannot provide identity and so they must inherit it from some other type. In other words, these types are used to model their ancestors' attributes in certain points in time.

Role is used to model the fact that its ancestor may play a certain role in relation to another object. That object is connected to the role using an association. This relation is essential as role without the related object does not make sense (e.g. a Person may play a role of an Employee in relation to a Company, but without the Company the role is nonsensical).

Phases are used to represent different states of their ancestor. The ancestor may freely change the phase it is in, but can only be in exactly one of the phases. This is asserted by the fact, that the related phases must form a *phase partition* i.e. a disjoint and complete generalization set that represents all the different phases the ancestor can be in.

2.1.3 Category, Role Mixin and Mixin

As non-sortal types do not provide identity they cannot be directly instantiated. Instead they are used to group other types. This is useful for simplification and removal of duplicities in the model.

Of three non-sortal types, Category is the only one that is rigid. It defines a set of attributes common to the Kinds and/or Subkinds that inherit from it.

Role Mixin is antirigid type used to group different Roles together. Another way to look at Role Mixins is that they are a kind of an abstract Role.

Mixin is semi-rigid meaning it groups rigid and non-rigid types. To achieve semi-rigidity they must have at least one rigid and at least one non-rigid type as their descendants.

2.1.4 Aspects

Aspects are types of entities that are existentially dependent on another type (called *aspect bearer*). This means instance of an aspect cannot exist without its bearer. We use aspects to model properties that are structured or otherwise too complex to be modelled using attributes. There are four types of aspects:

- Perceivable Quality
- Non-Perceivable Quality
- Nominal Quality
- Mode

Qualities represent such properties of the bearer that can be quantified by some value. Perceivable Quality represents a property the value of which can be measured (e.g. size, color, etc.). Non-Perceivable Quality on the other hand cannot be measured (e.g. currency). Lastly, Nominal Quality represents a property that is not intrinsic to the bearer but has been assigned to it to make reference to it (e.g. serial number). Mode is used for properties that cannot be quantified (e.g. the edition of a book, or a skill of an individual).

2.2 Relations

Relations represent ways entities can interact with each other or be composed to form more complex ones. OntoUML introduces a precise system of relation types and meta-attributes that will be described in the following subsections.

2.2.1 Formal and material relations

Formal and material relations expand upon UML's association. Formal relation is such a relation between entities that is dependent solely on their intrinsic properties (e.g. 'taller than').

Material relations in contrast require an external entity (a *truthmaker*) to make them valid[2]. OntoUML defines a special entity type for this – Relator. The sides of the relation are connected to the Relator using a special mediation

[2]The truthmaker does not necessarily have to be a physical thing like a contract, it can also be something like a consensus or a deal that has no representation in the physical world.

relation. There can also be material relations derived from a Relator to specify the name of the relationship between the relation sides directly.

2.2.2 Part–whole relations

OntoUML puts great emphasis on part–whole relations. It uses the UML's symbols for aggregation and composition (filled and blank diamonds), but redefines their meaning. Blank diamond means that the part can be shared among different wholes whereas the filled one means that the part may belong to at most one whole at a time. OntoUML also introduces various meta-attributes of the part–whole relations used to further specify their semantics:

- Mandatory part – the aggregating object must contain at least the specified minimum of the parts (e.g. a living person must have a heart to live), this is modelled by setting the lower bound on cardinality on the part's side to a non-zero value
- Essential part – the aggregating object is existentially dependent on the part and the part contributes to its identity (e.g. the relation between an art collection and its artifices), the `essential` meta-attribute is used to express this
- Optional whole – the part can exist without being aggregated (e.g. a light bulb can exist without being attached to a lamp), in this case the lower bound on cardinality on the aggregating object's side is zero
- Mandatory whole – the part has to be contained in an aggregating object, this fact is declared with non-zero cardinality on the aggregating object's side
- Inseparable part – the part cannot be removed from the aggregating object without ceasing to exist (e.g. we cannot remove a hole from a shirt without 'destroying' it), there is the `inseparable` meta-attribute for this situation
- Inseparable essential part – the part cannot exist without the aggregating object and vice versa (e.g. a person and their brain), both `essential` and `inseparable` meta-attributes are used in this case

There are two more meta-attributes that aim to cope with the non-rigid types' ability to change over time:

- Immutable part – the part must stay the same independently on the aggregating object's current state – the `immutable part` meta-attribute
- Immutable whole – analogous to the previous one with the part and the aggregating object exchanged, the meta-attribute is `immutable whole`

Three types of aggregating objects are distinguished in OntoUML:

1. Quantity
2. Collective
3. Functional whole

We use *Quantity* (a special entity type that provides identity) to model scenarios where the part is infinitely divisible and is defined by its topology. This means that every Quantity must have a relation to its container that defines the topology (either directly or via another Quantity it composes). This relation has the *containment* stereotype. Quantities can also be composed from other Quantities with a special *subQuantityOf* relation.

Collectives are used to model groups of other objects that share the same role in that group (the role defines what is called the *unifying relation* of the collective). There are two relation types dedicated to Collectives – *memberOf* used to assign members to a Collective and *subCollectionOf* to compose a Collective from other Collectives.

Functional whole is the most basic of aggregating object – it is composed of different parts that have different purpose in it. These parts are assigned to the aggregating object using the *componentOf* relation.

Chapter 3

Object Model

Object model is the set of basic concepts upon which object-oriented programming is built. There are only a few of them [21]:

Object An entity that represents some part of the system. It has inner state represented by its set of *attributes* (sometimes called instance variables) and its *protocol* – set of messages sent by other objects using its *methods* it can receive.

Class As defined in [22], classes are 'extensible templates for creating objects, providing initial values for instance variables and the bodies for methods'. Objects created using a class are called its *instances*.

Constructor A special method of a class that creates new instances. In many object-oriented languages there can be more than one constructor for a class allowing for different initialization for different use cases.

Collection A special object that can hold references to a number of other objects. In pure object-oriented languages (e.g. Smalltalk) members of the collection can be of different types (those collections are called heterogeneous), in other languages (such as C++, Java or C#) members need to have the same type (or at least a common ancestor type).

There are two main types of relations between classes:

Inheritance the relation between a class and a class that is its specialization (also called *is-a* relation). The specialized class usually overrides or extends the behaviour of its parent. A class can have multiple parents at the same time (although in some object-oriented languages the number of parents is limited to one, e.g. in C#).

Aggregation the relation between a whole and a part – the aggregating object contains aggregated objects as its parts.

Chapter 4

C#

C# is a type-safe, garbage-collected, compiled, object-oriented [3] language for Microsoft's .NET platform [23]. Its first version was published in 2002 and the latest one at the time of this thesis (6.0) in 2015. In this section we will discuss how it copes with implementing the Object model while first describing its part relevant to this topic.

4.1 Type system

C# is a type-safe language, meaning every object is an instance of some type. There are two main kinds of types: value types and reference types. Value types represent value-passed types of objects (e.g. numbers, enums or structs). Instances of reference types are passed by reference (e.g. classes, arrays and interfaces).

There is a special literal — `null` — that represents a so called null reference — reference that does not point to any object. Any variable of a reference type can hold the `null` value. To be able to have instances of value types that can also have the value `null`, so called *Nullable types* were introduced to C# (see section 4.1.10 of [23]). These are denoted by the `?` suffix (e.g. nullable `int` is `int?`).

For the purpose of discussing the Object model implementation, we will focus on classes as they offer the functionality most relevant to that.

4.2 Members

Classes may contain two basic types of *members* – *static* and *instance*. Static members are those that belong to the class itself (denoted using the keyword `static`) whereas instance members belong to the individual instances of the

[3] There are also aspects from functional, component-oriented, and other programming paradigms but for this thesis, we will cover only the object-oriented side of C#.

class. Classes can be marked `static` meaning they cannot have instances. Members can be of several types, here are those relevant to this thesis [23]:

- Constants – named constant values
- Fields – named attributes of the class
- Methods – actions the class can perform
- Properties – special actions used to read and/or write a named property of the class (typically a field or a computed value)
- Constructors – actions to initialize instances

Constants, fields and properties can be of any type. The fact that constants are not modified is checked at compile time.

Methods must specify their return type (there is a special return type `void` for methods that do not return a value) and can take parameters. They can also be overloaded i.e. there can be more than one method with the same name but different set of parameters. Methods can be marked as `virtual` making them late bound, or `abstract` i.e. virtual with no implementation. Virtual (and of course also abstract) methods can be overridden in derived classes using the `override` keyword.

Properties are a special way to define a member that is similar to a field but can have actions associated with reading and/or writing to it. The read and write (called *get* and *set*) parts can be marked as `private` or `protected` independently and can also be marked as `virtual` or `abstract`. Similarly to methods, virtual and abstract properties can be overridden.

Constructors provide a way to initialize other members of an instance (or the class itself if the constructor is static). Instance constructors can be overloaded similarly to methods. Constructors can also include the call to base class' constructor (see section 4.3) that is called before the current constructor. This is denoted by a `base` call between the argument list and the constructor body (see section 10.11.1 of [23] for more information).

4.2.1 Access modifiers

In C# members can have five different levels of visibility to other parts of the program [24]:

- public – the item is accessible without restriction
- internal – the item is accessible only within the defining *assembly* (roughly equivalent to Java's jar archives)
- protected – the item is accessible only to the defining class and its instances or its subclasses

- protected internal – a union of protected and internal, the item is accessible via defining class and subclasses *or* within the defining assembly
- private – the item is accessible only to the defining class and its instances

4.3 Inheritance and interfaces

Inheritance is implemented in a way that allows exactly one predecessor class called 'base class' to be set. If no base class is explicitly specified, special `object` class is used instead[4]). Classes can be marked with the `sealed` keyword to indicate that no other class can inherit from them.

Classes can implement multiple interfaces – contracts that specify a set of methods and/or properties the implementing class contains. Classes can also be *generic*, meaning that types of some of its members can be parametrized. Various constraints can be defined for the type parameters.

As stated in section 4.2, class members can be marked as `virtual` which allows child classes to override their behaviour (using the `override` keyword), see section 1.6.6.4 of [23]. Also even when the member is not marked as `virtual`, the derived class can *hide* it using the `new` keyword (see section 3.7.1.2 of [23]).

4.4 Namespaces

Namespaces represent a way to organize the structure of C# programs. Using those, classes can be grouped into logical chunks to make the structure of the program clearer. Reference to a namespace is declared by the `using` statement (e.g. `using System;`) usually at the top of a file. See section 9 of [23] for more.

4.5 Syntactic features in C# 6.0

The latest version of the C# language introduces some useful syntactic constructs that help to make the code shorter and more to the point (as there is no specification for this version yet, I suggest reading [25] for more info). In this thesis, I use three of them: null-propagating operator, expression bodied members and `nameof` expression. These are all compile-time features, so to use them, Visual Studio 2015 or newer is required.

The null-propagating operator `?.` (often called 'elvis operator') facilitates navigating through members that might be null. In listing 4.1 we see an example. The assignment to `age1` ends with an instance of `NullReferenceException` being thrown. Checking whether every needed member is not null is quite

[4] `object` is therefore the only class that does not have a predecessor.

tedious and so in C# 6.0 the null-propagating operator can be used as seen in the assignment to `age2`. If either `a` or `Age` is null the value assigned to `age2` is null and no exception is thrown.

```
class Person { public int? Age { get; set; } }
Person a = null;
var age1 = a.Age.ToString(); // NullReferenceException
var age2 = a?.Age?.ToString(); // OK, age2 == null
```

Listing 4.1: Null-propagating operator example

Expression bodied members provide a way to simplify single expression methods and get-only properties declaration. In listing 4.2 we can see this clearly. Functionally equivalent `SayHi` and `SayHiNew` represent the method simplification, `Initial` and `InitialNew` the get-only properties.

```
class Person
{
  public string Name { get; set; }

  public string SayHi(string salute)
  {
    return salute + ", " + Name;
  }

  public string SayHiNew(string salute)
    => salute + ", " + Name;

  public char Initial { get { return Name[0]; } }

  public char InitialNew => Name[0];
}
```

Listing 4.2: Expression bodied members example

The `nameof` expression is used to get names of members in a type checked way. Every `nameof` expression is resolved to a string at compile time. Example can be seen in listing 4.3. The advantage of using `nameof` over string constants is that when the argument of `nameof` is renamed and we forget to update it, the build fails and we notice the error easily. This is useful for example when passing the name of the parameter that is null to the `ArgumentNullException` constructor (it expects the name as a string).

```
class Person { public int Age { get; set; } }
var nameOfPerson = nameof(Person); // "Person"
var p = new Person();
var nameOfAge = nameof(p.Age); // "Age"
```

Listing 4.3: nameof example

4.6 C# and the Object Model

C# does not precisely implement the Object Model described in chapter 3. All the Object Model aspects are enumerated once again and it is discussed how C# deals with them below:

Object Instances in C# conform to the definition without exception – they have inner state represented by the values of their fields and they can expose a set of methods creating their protocol.

Class Classes align with the definition as well – they provide the template for creating objects and can be extended by inheritance. Classes are not a first-class object in C#, they must be handled by a special *Type* class instances.

Constructor Constructors in C#behave as stated in the Object Model definition and can be overloaded.

Collection Collections are homogeneous in C# meaning they can only contain instances of one particular type.

Inheritance In C#a class can only have one other class as its ancestor (and multiple interfaces). This will prove to be quite a challenge to overcome (see section 8.3.2.2).

Aggregation Objects can hold references to other objects in C#, so this is implemented with no dissonance from the Object Model

Chapter 5

PIM languages

In this chapter we will discuss existing languages suitable for using as Platform Independent Model in OntoUML into source code transformation process.

5.1 RDF, RDF Schema and OWL

RDF (Resource Description Framework) is a framework for expressing information about resources[26] (resource meaning anything from concrete individuals to abstract concepts). It was created by W3C as standardized means to represent various metadata of web content used for example during machine processing. As such, RDF is meant to be used as a base for other semantic extending notations.

For data modelling RDF was extended by RDF Schema (often abbreviated as RDF-S). This extension adds concepts like classes and properties with semantics very similar to object-oriented programming languages (including inheritance and so on).

OWL (Web Ontology Language) is an ontology language for semantic web. It is built on RDF and RDF-S further enriching the RDF-S vocabulary with tools that enable us to very precisely describe relations between entities[27]. The current version is OWL 2 released in 2012.

OWL is also used to query over models (commonly used term for this is *reasoning* with the model). There are three so called profiles of OWL 2: OWL EL, OWL QL and OWL RL. These represent subsets of OWL constrained in certain ways to guarantee certain levels of computational complexity of the reasoning. As we are not going to reason with our models, further details of those profiles fall out of the scope of this thesis (specification can be found in [28]) and from now on we will consider OWL 2 in its unconstrained form.

OWL contains constructs that are very similar to those used in OntoUML. For example a class can be an intersection and/or union of other classes which is close to OntoUML's interpretation of generalization sets.

5.1.1 Code generators

There are several code generators that are able to generate Java or C# code from OWL models:

- ROWLEX – C# generator, the project seems discontinued from at least 2012[29]
- Jastor – Java generator[30], latest version (1.0.4) was released in 2006[31]
- protégé – OWL editor capable of generating Java code[32], this project is still active, latest version was released in late 2015[33]

As we can see, although OWL is a standard language, code generation support is quite problematic as most of the projects related to this topic are defunct and their support has ended.

5.2 RefOntoUML

RefOntoUML is a language developed by NEMO (Ontology & Conceptual Modeling Research Group), a research group around the author of OntoUML, for their *OLED* editor [8]. It is a simple XML based language describing the OntoUML model. The structure closely aligns with OntoUML structure and as such resembles more or less a serialization of the OLED's internal representation. This means the resulting file is not very human readable and more importantly due to its ad hoc nature is not very well known or supported by other tools. *Menthor*, the closed-source spin-off of OLED[34], supports the import of RefOntoUML files from OLED, but exports RefOntoUML files that are slightly different. For the purpose of this thesis we will consider only the files generated by OLED.

5.2.1 Code generators

As mentioned above, RefOntoUML is a language created for a concrete tool. At the time of writing this thesis, the latest stable version of OLED is version 2.0.1 [35]. There are no functions that allow for generation of any object-oriented code (there is an OWL generator, however). The same stands for Menthor.

Part II

Solution

Chapter 6

Solution overview

The solution consists of a set of transformations that gradually turn OntoUML model into target language source code and the application that provides a framework for these transformations and illustrates this process by transforming RefOntoUml files into C# classes.

6.1 Transformations

To allow for maximum extensibility of the program the transformation is divided into four subtransformations as shown in Figure 6.1 (the diagram uses Yourdon Data Flow Diagram notation [10]). All of the white transformers can be added to either support another input form or different output language. The greyed out transformation is the core in which the transformation from OntoUML as a Computation Independent Model (CIM) is transformed into OntoObjectModel – a Platform Independent Model (PIM) – and is not to be extended.

6.2 Technologies used

The application was written on Node.js platform in the TypeScript language. The reason I chose this combination was to be as multiplatform as possible (Node.js) and to have the comfort of type safety (TypeScript). Also it allows for easy prototyping of the concepts I needed to show in this thesis and moreover, I have been working with this stack for quite a while so I have some experience with it.

There emerged a need for some helper C# classes to facilitate the work with associations and collections with bounds on the number of items from the generated C# code. Therefore I created a NuGet package with those (see section 6.4). In this section all the technologies used are briefly introduced.

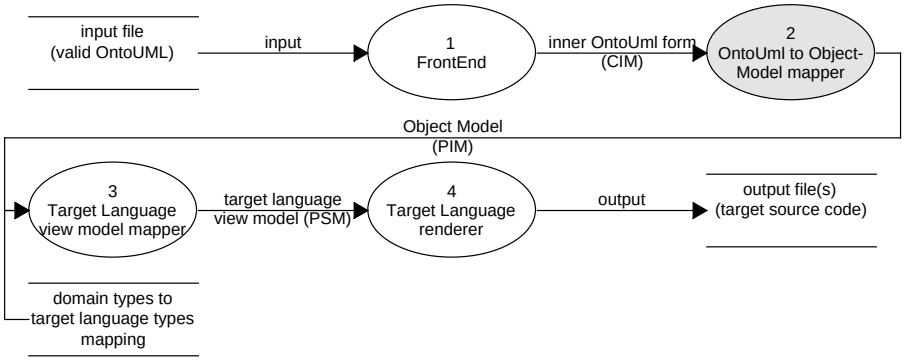


Figure 6.1: Transformation sequence

6.2.1 Node.js and npm

Node.js is 'JavaScript runtime built on Chrome's V8 JavaScript engine' ([2]). It is used to build a wide range of applications from web servers to command-line utilities. The runtime can be run on Windows, Mac OS X, Linux and several other platforms [36].

Applications and libraries written in Node.js can be distributed as modules. The official package manager system is called *npm*. At the time of writing this thesis, there were more than 250,000 public open-source packages available on npm which is almost twice as much as on Maven Central (Java package registry) and five times as much as on NuGet Gallery (C# package registry, see section 6.2.6) [37].

6.2.2 NPM packages used

As stated before, there are many useful npm packages out there. The application uses several of them to perform various tasks:

- `command-line-args` – package to handle command line arguments parsing and to generate usage guide for the application [38]
- `handlebars` – JavaScript templating engine [39]
- `json-format` – simple JSON formatter [40]
- `lodash` – utility library providing tools to facilitate functional programming and many other functions [41]
- `mkdirp` – utility to recursively create directories (similar to `mkdir -p` command in Unix systems) [42]

- `q` – library implementing JavaScript Promises (see section 6.2.5) [43]
- `tv4` – library for JSON Schema validation (see section 6.2.4) [44]
- `xml2js` – library for simple XML to JavaScript objects conversion [45]

The usage of each package will be mentioned in the relevant sections. There are also so called *devDependencies* – these are packages used to build and test the application (not while the application itself is run):

- `chai` – assertion library for unit testing [46]
- `chai-as-promised` – plug-in for `chai` that facilitates writing assertions for Promises [47]
- `istanbul` – code coverage library for JavaScript [48]
- `mocha` – unit test framework [49]
- `mock-fs` – library for mocking the calls to file system [50]
- `remap-istanbul` – library for remapping `istanbul` JavaScript results to TypeScript source files [51]
- `typescript` – official TypeScript compiler for Node.js [52]
- `typings` – type definition manager for TypeScript [53]

6.2.3 TypeScript and Typings

TypeScript is an open source 'typed superset of JavaScript that compiles to plain JavaScript' ([1]). It was created by Microsoft and is still actively developed. It adds static typing to JavaScript that helps to prevent errors, as well as some other useful features from the modern ES2015 (formerly called ES6) JavaScript specification [54] (such as arrow functions, destructuring and others) that can be transpiled to the more commonly supported ES5 version.

As TypeScript is a strict superset of JavaScript, all valid JavaScript sources are also valid TypeScript sources. This means existing JavaScript libraries (and therefore npm packages) can be used in TypeScript. To interoperate with them it is very beneficial (although not necessary) to include so called *type definitions file* for the used library. This file contains all the type information about the library allowing TypeScript to perform type checking when using them. There is a popular utility called Typings [53] (that obsoleted the previously used TSD utility) that allows users to search for, install and update type definitions for the libraries they want to use. This was utilized for several npm packages that are used by the application and for which the type definitions file existed at the time.

6.2.4 JSON Schema

JSON Schema is a way to describe the structure of JSON documents [55]. It can be viewed as an analogy of XML Schema (XSD) and XML files. It is itself a JSON based document that can also be (and actually is) described by a JSON Schema. It is used to validate all the JSON inputs to the application – serialized OntoObjectModel (see section 7.3.1) and PrimitiveTypeMapping files (see section 9.2).

6.2.5 Promises

In Node.js many operations are asynchronous (e.g. file I/O) and are handled via providing callback function to be called when the operation is completed. This approach is simple but can lead to very deeply nested code in situations when another asynchronous operation is called from within a callback. This is one of the reasons I decided to use Promises.

Promise is a special type of object that allows 'to associate handlers to an asynchronous action's eventual success value or failure reason' ([56]). Promises also support chaining i.e. a Promise's success handler can return another Promise etc. This brings some advantages to the table, along with the improved readability of the code, they allow to write nicer interfaces (see section 10.1 for an example) and to centralize error handling (we can specify one 'catch-all' error handler instead of writing one for every asynchronous operation). Although Promises are natively supported in Node.js, there are still some issues with it (e.g. [57]), so I chose to use the `q` library for it (see section 6.2.2).

6.2.6 NuGet

According to its official website 'NuGet is the package manager for the Microsoft development platform including .NET. ' ([5]). It enables developers to easily manage external packages for .NET projects (libraries etc.), their dependencies, updates and so on. It also abstracts the developer from integrating the library in the package into their project (setting up references and other things).

6.3 Application

In this section the application created for this thesis is described. First its architecture is overviewed and then instructions on how to run it are provided. The application sources can be found on the enclosed DVD and also on [58].

6.3.1 Application architecture

The application architecture closely aligns with the transformation flow depicted in fig. 6.1. The application entry-point – `app.ts` parses the command-line

arguments (using the `command-line-args` package, see section 6.2.2) and according to their values chooses the front-end implementation, output language mapper and renderers (for argument reference see section 6.3.3). After that, front-end's `parseFile` method is called (see section 7.1) and on the resulting Promise (see section 6.2.5), subsequent calls are chained. Those calls are in order:

1. `OntoUmlToOntoObjectModelMapper` call, unless the front-end has its `returnsOntoObjectModel` flag set to true indicating it returns OntoObjectModel directly (see chapter 8)
2. the selected target language mapper's `modelToViewModel` call (chapter 9)
3. all the selected target language renderers' `generateCode` calls (chapter 10)

6.3.2 Running the Application

The only requirement to run the application is to have the Node.js runtime of version at least 5.x installed (which also installs the npm utility 3.x along with itself). It can be downloaded from [36]. The application is initialized using the `npm install` command. This ensures that all of the packages depended upon are installed (see section 6.2.2), type definition files are downloaded (see section 6.2.3) and the TypeScript files are compiled into JavaScript. Unit tests can be run using the `npm test` command and test coverage report can be generated using the `npm run code-coverage` command. This command creates a HTML page in the *coverage* directory where details on unit test code coverage can be found. The application can be run using the `npm start` command, additional parameters can be added like this: `npm start -- --parameter value` (note the double dash between `start` and the parameters). See section 6.3.3 for parameter reference.

6.3.3 Application parameters

The application can be run with several command-line parameters that can be used to adjust the code generation. They are listed bellow in the form usual in command-line utilities' help (the same text can be also obtained by running the application without parameters or with the `--help` parameter):

`--input, -i` – Path to the input file. If a parameter is not prefixed with any name, it is assumed that it is this one (i.e. `npm start -- -i file.txt` is identical to `npm start -- file.txt`).

`--inputForm, -I` – The input form code. Valid values are (default is `refontouml`):

`refontouml` – RefOntoUml from OLED (see section 5.2).

`onto-object-model` – JSON of the OntoObjectModel (section 8.2).

`--output, -o` – The path to the output directory (or if `--singleFile` flag is set the name of the output file). If it does not exist, it is created.

`--outputForm, -O` – The output form code. Valid values are (default is `csharp-model`):

`csharp-model` – C# model classes (see section 9.3).
Config options:

`-c namespace=<namespaceName>` – Name of the namespace to use in the generated file(s) (see section 9.3.1).

`onto-object-model` – JSON of the OntoObjectModel.

`--typeMapping, -t` – Path to the type mapping file (see section 9.2).

`--singleFile, -s` – If set, all the generated code will be placed in a single file.

`--config, -c` – Additional configuration options in the form `<name>=<value>`.

`--help, -h` – Displays the help message with usage information and parameter reference and quits.

`--verbose, -v` – Displays additional output (e.g. generated files' content).

6.4 C# utilities library

To address some of the conceptual constraints from OntoUML in the generated code I created a NuGet package `Ccmi.OntoUml.Utilities` (for more information on NuGet see 6.2.6). It consists of two main parts – classes to work with associations and collections that hard check the bounds on the number of items they can contain. The package source files are available on the DVD enclosed to this thesis and also on [59]. The built package is accessible from the feed in [60].

6.4.1 Association classes

In the `Ccmi.OntoUml.Utilities.AssociationClasses` namespace, there are three classes, one for each of the association types:

- `OneToOneAssociation<TLeft, TRight>` for one to one associations
- `OneToManyAssociation<TOne, TMany>` for one to many associations
- `ManyToManyAssociation<TM, TN>` for many to many associations

The implementation is an adapted port of the Java version proposed in [61]. The main advantage of this approach is that both ends can navigate to each other without synchronization issues that are introduced when implementing associations naïvely using attributes.

The singleton implementation utilizes the `System.Lazy<T>` class[62] in `OneToOneAssociation<TLeft, TRight>` and standard double checked locking to prevent race conditions in the other two. `Lazy<T>` could not be used in those because of the constructor parameter[5]. All three classes have similar interfaces:

- Instance accessor taking optional parameter that specifies whether there can be multiple associations between the same instances (this parameter is obviously missing in one to one association). The parameter defaults to false meaning there cannot be multiple associations between the same instances.
- Methods to create association between an instance of one type and many instances of the other type (except for the 1:1 where there can be only one). It throws an instance of `System.InvalidOperationException` if duplicate associations are not allowed and an attempt to add such association is made.
- Method to remove an association between two objects. If duplicate associations are allowed, only one is removed. If no association between the objects is specified, the method does nothing.
- Methods to retrieve associated objects for a specified object. If there can possibly be only one result and none is found, the method returns null, if there can be more than one, it returns an empty `Enumerable<T>` of the appropriate type.

6.4.2 Bounded collections

The `Ccmi.OntoUml.Utilities.Collections` namespace contains interfaces and implementation of collection that checks the number of items in it against provided bounds.

There are two interfaces (`IBoundedCollection` and `IBoundedList`) that are similar to their `System.Collections.Generic` counterparts [63] and the `BoundedList` that implements the `IBoundedList`. The implementation is based on inheriting the `System.Collections.Generic.List<T>`[64] and hiding those of its methods that can change the number of items in it using the `new` construct (see section 4.3). When a bound is about to be crossed, instance of `System.InvalidOperationException` is thrown. The bounds can be set using the constructor that has two optional parameters for upper and lower bound of the items count (`maxItems` and `minItems` respectively) or setting the appropriate property (`MaxItems` and `MinItems`) – these assert that the new

[5]This could have been circumvented by creating two classes per association type (e.g. `ManyToManyAssociation<TM, TN>` and `UniqueManyToManyAssociation<TM, TN>` but this would lead to code replication across those which I found undesirable.

bound is compatible with the current number of items (e.g. not trying to set the upper bound to 5 on an instance that already contains 10 items) throwing `InvalidOperationException` if the assigned value is not valid.

Chapter 7

Input form into OntoUML transformation

To be able to abstract the core transformation from OntoUML into OntoObjectModel from the input format, there must be a layer between the input files and that transformation. This chapter describes how this layer is constructed and discusses the concrete implementation.

7.1 Purpose and interface

The purpose of this layer (taking analogy in compilers, I call it *front-end*) is to read the input file and transform it into the inner OntoUML representation. Its interface is quite simple:

```
interface IFrontEnd {
  parseFile: (file: string) =>
    Q.Promise<OntoUmlModel | OntoObjectModel>;
  returnsOntoObjectModel: boolean;
}
```

Listing 7.1: Front-End interface

The `parseFile` method takes the path to the input file as a parameter and returns a promise to the result of the transformation into either OntoUmlModel or directly into OntoObjectModel (effectively bypassing the core transformation to allow inputting for example OntoObjectModel serialized to a file). Whether OntoUmlModel or OntoObjectModel is returned is specified by the `returnsOntoObjectModel` flag.

7.2 Inner OntoUML representation

The inner OntoUML form is represented by interfaces in the `OntoUml` module. Its structure is defined in this section.

7.2.1 OntoUmlModel

The entire model is encapsulated in an object of `OntoUmlModel` type. This object contains three dictionaries:

- `entities` – all the entities in the model (instances of `OntoUmlEntity`, see section 7.2.2)
- `generalizationSets` – all the generalization sets in the model (instances of `OntoUmlGeneralizationSet`, see section 7.2.5)
- `relations` – all the relations between entities in the model except for generalizations (`OntoUmlRelation` instances, see section 7.2.6)

All these dictionaries are keyed by the name of the items in them. Those are considered to be unique across the model.

7.2.2 OntoUmlEntity

Each entity (a class) in the OntoUML model is described by an object of the `OntoUmlEntity` type. It has the following properties:

- `name` – unique name of the entity
- `type` – OntoUML object type of the entity (e.g. Kind, Role, etc.) specified by an `OntoUmlEntityType` enum value
- `attributes` – all the attributes of the entity (`OntoUmlAttribute` instances, see section 7.2.3)
- `generalizations` – all the generalizations in the model where this entity is a successor (`OntoUmlGeneralization` instances, see section 7.2.4)

7.2.3 OntoUmlAttribute

Every attribute of an entity is described by an object of the `OntoUmlAttribute` type. It contains these properties:

- `name` – unique name of the attribute
- `type` – name of the domain type of the attribute
- `minItems` – minimal count of items in the attribute (lower bound on its cardinality)

- `maxItems` – maximal count of items in the attribute (upper bound on its cardinality), −1 for unlimited

7.2.4 OntoUmlGeneralization

The generalization relation of an entity and its predecessor entity is represented by the `OntoUmlGeneralization` type. It has two properties:

- `predecessor` – name of the predecessor entity
- `generalizationSet` – (optional) name of the generalization set this generalization contributes to

7.2.5 OntoUmlGeneralizationSet

`OntoUmlGeneralizationSet` instances are used to describe all the generalization sets in the model. Its properties are:

- `name` – unique name of the generalization set
- `childrenNames` – names of the children entities
- `isComplete` – flag indicating whether the generalization set is complete (see section 2.1)
- `isDisjoint` – flag indicating whether the generalization set is disjoint (see section 2.1)

7.2.6 OntoUmlRelation

Relations between entities are described by the `OntoUmlRelation` type. It has several properties:

- `name` – unique name of the relation
- `type` – OntoUML relation type (e.g. Material, MemberOf, etc.) specified by an `OntoUmlRelationType` enum value
- `sourceEnd` – source end of the relation (instance of `OntoUmlRelationEnd`, see section 7.2.7)
- `targetEnd` – target end of the relation (instance of `OntoUmlRelationEnd`, see section 7.2.7)
- `isShareable`, `isImmutablePart`, `isImmutableWhole`, `isEssential`, `isInseparable` – flags indicating whether the respective part–whole relation meta-attributes are present (see section 2.2.2)

- `allowDuplicates` – flag indicating that the relation can exist multiple time between the same instances
- `derivedFrom` – name of the relator this relation is derived from (see section 2.2.1

7.2.7 OntoUmlRelationEnd

`OntoUmlRelationEnd` describes an end of a relation. It has properties analogous to `OntoUmlAttribute`:

- `name` – unique name of the end field
- `type` – name of the entity this end represents
- `minItems` – minimal count of items in the relation end (lower bound on its cardinality)
- `maxItems` – maximal count of items in the relation end (upper bound on its cardinality), -1 for unlimited

7.3 Implementation

To illustrate the extensibility of the front-end side, two front-ends have been implemented: *OntoObjectModelFrontEnd* and *RefOntoUmlFrontEnd*.

7.3.1 OntoObjectModelFrontEnd

OntoObjectModelFrontEnd allows to parse OntoObjectModel serialized to JSON and bypasses the core transformation. It is useful for debugging the final code generation and as such was the first front-end I implemented to test the technologies and techniques I used in those stages of transformation. It is extremely simple – it just reads the file specified, validates that the file is a valid OntoObjectModel using JSON Schema and the `tv4` package, calls the internal `JSON.parse` method and returns a promise to the result.

7.3.2 RefOntoUmlFrontEnd

RefOntoUmlFrontEnd parses RefOntoUml files from OLED (see section 5.2) and returns promise to an OntoUmlModel. Upon closer inspection we can see that the RefOntoUml file structure is very similar to the OntoUmlModel structure. This in fact was one of the reasons I chose RefOntoUml (the other one being that OLED is one of the best OntoUML editors I know).

It uses the `xml2js` package (see section 6.2.2) to easily transform XML-based RefOntoUml files into JavaSript objects. After the XML is parsed, the front-end does some fairly simple transformations such as mapping the entity

and relation types to enum values, converts strings to numbers or boolean values where applicable and structures list-like XML attributes to arrays. As the transformation is so simple, it will not be described here.

Chapter 8

OntoUML into OntoObjectModel transformation

This chapter describes the core transformation from OntoUML model into OntoObjectModel. During this step, most of the OntoUML concepts are translated to object model constructs. The result of this transformation can then be used to render the output source code (see sections 9 and 10).

8.1 Purpose and interface

This transformation is designed to take OntoUML model — a CIM — as an input (see section 7.2) and implement many of its aspects using OntoObjectModel — a PIM — as a result (see section 8.2). Its interface consists of a single method:

```
interface IOntoUmlToOntoObjectModelMapper {
  mapToOntoObjectModel: (model: OntoUmlModel) =>
    OntoObjectModel;
}
```

Listing 8.1: OntoUmlToOntoObjectModelMapper interface

The `mapToOntoObjectModel` method takes an OntoUmlModel instance as its only argument and returns an OntoObjectModel instance. All the underlying transformations are hidden in the `OntoUmlToOntoObjectModelMapper` instance.

8.2 Object Model representation

This section covers the Object Model representation – OntoObjectModel. It is a JSON format that can be validated using the provided JSON Schema (see ODKAZ).

8.2.1 OntoObjectModel

The whole OntoObjectModel is represented by an instance of the eponymous type. It has the following properties:

- `classes` – all the classes in the model, also covering the inheritance structure (`ClassInfo` instnaces, see section 8.2.2)
- `relations` – all the relations between the classes, except for inheritance (`RelationInfo` instances, see section 8.2.7)

8.2.2 ClassInfo

Every class in the OntoObjectModel is represented by an instance of `ClassInfo` type. The suffix *Info* was used to avoid confusion between the name `Class` and the `class` keyword in TypeScript (and many other languages). Therefore all the types in OntoObjectModel have this suffix for the naming to remain consistent. Its properties are listed below:

- `name` – unique name of the class, it is used as an identifier of the class in other properties (e.g. `superClass` property, see below)
- `attributes` – all the attributes of the class (`AttributeInfo` instances, see section 8.2.3)
- `methods` – all the methods of the class (`MethodInfo` instances, see section 8.2.5)
- `superClass` – (optional) name of the super-class of this class, see section 8.3.2.2 for more
- `unionClasses` – (optional) names of the classes this class is an union of, see section 8.3.2.2 for more
- `implementing` – (optional) names of the interfaces this class implements, see sections 8.3.4 and 8.3.8 for usage
- `isAbstract` – (optional) flag indicating the class should not be instantiated directly
- `isInterface` – (optional) flag indicating this class represents an interface i.e. has no method bodies

- `existentiallyDependentOn` – (optional) name of the class the given class is existentially dependent on, see section 8.3.6 for more

The presence of the `methods` property may seem surprising as OntoUML does not support definition of methods. However, I wanted to take all the object model aspects into account, so methods are included and supported in the mappings (this can be tried by using the `OntoObjectModelFrontEnd`, see section 7.3.1).

8.2.3 AttributeInfo

Each attribute can be described using instances of `AttributeInfo` type. This type has the following properties:

- `name` – name of the attribute unique across the attributes of the same class
- `typeInfo` – (optional) information about the attribute type (`TypeInfo` instance, see section 8.2.4)
- `minItems` – (optional) minimal count of items in the attribute, defaults to 0
- `maxItems` – (optional) maximal count of items in the attribute, -1 for unlimited, defaults to 1

8.2.4 TypeInfo

Information about the type of an attribute or a method parameter is encapsulated into instances of the `TypeInfo` type. Its properties are as follows:

- `name` – name of the domain primitive type or a class in the model
- `isReference` – flag indicating the `name` field denotes a class this type, not a domain primitive type name

8.2.5 MethodInfo

Method signatures are described by `MethodInfo` instances [6]. They have these properties:

- `name` – name of the method
- `parameters` – (optional) description of all the method's parameters (`ParameterInfo` instances, see section 8.2.6)

[6] We are not concerned with method bodies here as they cannot be represented in platform independent way easily.

- `typeInfo` – (optional) the return type of the method (a `TypeInfo` instance, see section 8.2.4)

8.2.6 ParameterInfo

Method parameters are defined using instances of `ParameterInfo` type. It has three properties:

- `name` – name of the parameter,
- `typeInfo` – (optional) information about the parameter's type (`TypeInfo` instance, see section 8.2.4)
- `isCollection` – flag indicating the argument represents a collection of items

8.2.7 RelationInfo and RelationEndInfo

Relations and relation ends are represented in a very similar way as in OntoUmlModel (see sections 7.2.6 and 7.2.7) to convey the as much semantics as possible. There are only two differences: `RelationInfo` does not have the relation type and adds one attribute:

- `isPartInitializedWithWhole` – flag indicating that the part instance should be initialized at the same time as the whole (typically in the whole's constructor)

8.3 Implementation

This section covers the transformation from OntoUmlModel into OntoObjectModel (see sections 7.2.1 and 8.2.1 respectively). For clarity, the transformations are divided into thematically related subsections. The order of the transformation is as follows:

1. Basic mapping – creates a class for every entity and sets its attributes, super-class and union classes (see section 8.3.2)
2. Aspect mapping – handles the existential dependency property of aspect types (see section 8.3.6)
3. Overlapping inheritance mapping – creates the classes that are the result of overlapping generalization sets (see section 8.3.3)
4. Phase partitions mapping – creates auxiliary interfaces for each Phase partition and connects the Phases to their owner (see section 8.3.4)
5. Role mapping – connects all the Roles to their owners (see section 8.3.5)

6. Association-like relations mapping – maps the relations of certain types to their simplified versions (see section 8.3.7)
7. Special relations mapping – processes the SubQuantityOf, SubCollectionOf and MemberOf relations in a way that is similar to Phase partition mapping (see section 8.3.8)

8.3.1 Pseudocode notation

Pseudocode syntax used to describe the algorithms in this chapter and chapter 9 is loosely based on C syntax but is indentation based. The individual aspects are described below.

- `procedure` is used to denote a routine that does not return a value, `function` is for routines that do.
- The `if-then-else` and `while-do` constructs have the same semantics as in C or JavaScript.
- The `for all` $x \in y$ `do` construct iterates over the y collection and uses x to refer to the currently processed element.
- The '$\leftarrow$' symbol denotes assignment, all other logical and mathematical symbols retain their usual semantics.
- Curly braces are used to denote object literals, dots are used to access object attributes, *null* denotes a null reference.
- Square brackets are used to access collection members, [] means empty collection literal. Every collection has a `push` method used to add its argument to the end of the collection.

8.3.2 Basic mapping

Every entity in the OntoUmlModel is iterated over with the basic mapping. This creates a `ClassInfo` instance for every `OntoUmlEntity` instance in the model. The steps are:

1. The target's `name` is copied from the `name` property of the source
2. The primitive attributes are mapped, see section 8.3.2.1
3. Inheritance is mapped, see section 8.3.2.2

8.3.2.1 Attribute mapping

Primitive attribute mapping is straight-forward (see Algorithm 1). Most of the properties are copied and the type is wrapped into a `TypeInfo` instance with the `isReference` flag set accordingly.

Algorithm 1 Attribute mapping

```
1: function mapAttribute(attribute)
2:     return {
3:         name ← attribute.name,
4:         minItems ← attribute.minItems,
5:         maxItems ← attribute.maxItems
6:         typeInfo ← {name ← attribute.type, isReference ← ⊥}
7:     }
```

8.3.2.2 Inheritance mapping

To understand inheritance mapping, we need to discuss the difference between `superClass` and `unionClasses` properties of `ClassInfo`. The `superClass` property contains the name of the class this class inherits its identity from (see section 2.1) and there can be at most one. In the `unionClasses` the names of classes that the current class is an union of. This happens for example when creating classes that are the result of overlapping inheritance. The reason those two properties are not merged into one is that many OO languages do not allow multiple inheritance and so it is important to distinguish the 'true' super class from which the class should inherit. The algorithm to find the `superClass` and `unionClasses` is shown in Algorithm 2.

Algorithm 2 Getting the superClass and unionClasses

```
1: function getSuperClass(element)
2:     if element.generalizations is empty then return null
3:     candidates ← getPredecessorsFrom(element.generalizations)
4:     return findValidSuperClass(candidates, entity.type)
5: function getUnionClasses(element, superClass)
6:     if element.generalizations is empty then return []
7:     candidates ← getPredecessorsFrom(element.generalizations)
8:     return all c's such that c ∈ candidates ∧ ¬(c = superClass
9:             ∨isIdentityProvider(c))
```

The *getPredecessorsFrom* function projects the `generalizations` array to the `predecessor` property. The *findValidSuperClass* function filters the list of candidate predecessors according to the allowed type. These are shown in Table 8.1. For types not in the table the `superClass` is always null as those inheritances are handled in different ways. The *isIdentityProvider* function returns true if its argument is of type that provides identity – Kind, Subkind, Quantity or Collective (see section 2.1). Identity providers are excluded in the *getUnionClasses* function as they should be inherited via superClass.

Table 8.1: Valid super class types

Class type	Valid super class types
Subkind	Kind, Subkind
Role	Role
Phase	Phase

8.3.3 Overlapping inheritance mapping

When a generalization set is not marked as disjoint, there are classes needed for every combination of the subclasses (for distinction I will call them *atomic subclasses*). As the atomic subclasses are taken care of in another step (see section 8.3.2.2), this step handles only the classes that need to be created from the combinations (I call them *overlapping classes*). The algorithm is listed in Algorithm 3.

Algorithm 3 Generating overlapping classes

```
1:  procedure createOverlappingClasses(generalizationSet)
2:      if ¬generalizationSet.isDisjoint then return
3:      children ← getChildrenNames(generalizationSet)
4:      combinations ← getAllCombinations(children)
5:      for all combination ∈ combinations do
6:          names ← getNames(combination)
7:          addClass({
8:                  name ← concatAll(names),
9:                  superClass ← getSuperClass(combination[0]),
10:                 unionClasses ← names
11:         })
```

The *getChildrenNames* returns an array of all the children classes' names for a specified `OntoUmlGeneralizationSet`. The *getAllCombinations* returns all the combinations of the provided elements that have at least two members, the return form is an array of arrays. The *concatAll* function concatenates an array of strings passed to it into a single string. The *getSuperClass* function is the same as in Algorithm 2, here it is called with the first member of the combination (as all the members have the same superclass, it does not matter which one we use to determine the superclass of the overlapping class). The *addClass* function adds a new class to the mapping results.

8.3.4 Phase mapping

Because all the entities are mapped to classes by now, the mapping of phase partitions (see section 2.1.2) is realised altering the existing classes by these three steps:

1. adding an interface class for the phase partition
2. declaring that all the phase classes from the partition implement that interface
3. adding a relation between the class that can be in those phases and the partition interface

The relation is created in such a way, that Phase cannot separate itself from the owner and the owner must be initialized with a Phase. The process can be seen in Algorithm 4. There is an important precondition on the procedure – the generalization set must be disjoint and complete, otherwise it cannot represent a Phase partition and so calling this procedure is nonsensical.

Algorithm 4 Mapping of Phase partitions

Require: $generalizationSet.isDisjoint \wedge generalizationSet.isComplete$

procedure mapPhasePartition(*generalizationSet*)
  $children \leftarrow getChildren(generalizationSet)$
  if any of *children* is not a Phase **then return**
    ▷ All of the children in Phase partitions must be Phases
  $ownerClass \leftarrow getParent(generalizationSet)$
  $partitionName \leftarrow generalizationSet.name$
  $addClass(\{name \leftarrow partitionName, isInterface \leftarrow \top\})$ ▷ Step 1: add the interface
  for all $phase \in children$ **do**
    $phaseClass \leftarrow getClass(phase)$
    $phaseClass.implementing.push(partitionName)$
      ▷ Step 2: declare that Phase classes implement the interface
  $addRelation(\{$
    $name \leftarrow partitionName,$
    $isInseparable \leftarrow \top,$
    $isPartInitializedWithWhole \leftarrow \top,$
    $sourceEnd \leftarrow \{$
      $className \leftarrow ownerClass.name,$
      $name \leftarrow ownerClass.name,$
      $maxItems \leftarrow 1, minItems \leftarrow 1$
    $\},$
    $targetEnd \leftarrow \{$
      $className \leftarrow partitionName,$
      $name \leftarrow partitionName,$
      $maxItems \leftarrow 1, minItems \leftarrow 1$
    $\}$
  $\})$ ▷ Step 3: add the relation between owner and partition

The *getChildren* function returns all the child entities for a given instance of `OntoUmlGeneralizationSet`. The *getParent* function returns the parent

entity for a given `OntoUmlGeneralizationSet`. The *getClass* function returns the already mapped class that corresponds to the argument passed. The *addClass* function is the same as in section 8.3.3. The *addRelation* function adds a new relation to the mapping results.

8.3.5 Role mapping

Mapping of roles is very simple – as all the classes are mapped elsewhere, the only thing left to do is to add the relation between the class that plays the given role and the role itself. This approach is described in [21]. The relation is marked with the `isInseparable` flag, because when a role is separated from its owner it does not make sense anymore. The algorithm is shown in Algorithm 5.

Algorithm 5 Mapping of Roles

```
1: procedure MAPROLE(role)
2:     ownerClass ← getValidRoleOwnerClass(role)
3:     addRelation({
4:         name ← partitionName,
5:         isInseparable ← ⊤, allowDuplicates ← ⊤,
6:         sourceEnd ← {
7:             className ← ownerClass.name,
8:             name ← ownerClass.name,
9:             maxItems ← 1, minItems ← 1
10:        },
11:        targetEnd ← {
12:            className ← role.name,
13:            name ← role.name + "Role",
14:            maxItems ← −1, minItems ← 0
15:        }
16:    })
```

The *getValidRoleOwnerClass* iterates over its argument's generalizations and finds an ancestor that is a type that can play a role – Kind, Subkind, Quantity, Collective, Relator, Quality, Mode, Phase [65]. I excluded RoleMixin, because generalization between RoleMixin and Role does not mean that RoleMixin can play the Role. Also I set the upper bound of the roles collection to unlimited as there is no way to limit the number of roles an entity can play in OntoUML. The *addRelation* function is the same as in section 8.3.4.

8.3.6 Aspect mapping

The aspect types are existentially dependent on other types (see section 2.1.4 and we need to preserve this information in the class that represents the aspect.

The problem is that an aspect can be existentially dependent on another aspect and there we do not know which one is existentially dependent on the other just from the *characterization* relation. Therefore all the aspects in the model are processed at once by algoritm described in Algorithm 6.

The main idea is that every characterization hierarchy must form a tree with an identity provider in its root and every node in the tree is existentially dependent on its parent. Therefore the algorithm tries to find a non-aspect entity related to each aspect and if it finds one, it sets it as the existential dependency on the aspect. Otherwise, it tries to find a related aspect, that already has its dependency set (meaning it is closer to the root) and again if it finds one, it sets it as the dependency. If no dependency is found, the entity is pushed to the end of the processing queue in hopes that the other time around some of its related aspects will have been resolved and therefore will be able to be used as a dependency. This can be repeated at most the number of aspect types to the power of two times, as every time the whole queue is processed, at least one of the items in it must have been processed (either because it has a non-aspect related to itself, or one of its relatives has been resolved in the previous round). This limit is enforced to prevent infinite loops for invalid models.

The *Queue* and its methods behave like the standard 'first-in-first-out' queue. The *getCharacterizations* function return all the characterization relations in the model, that contain the entity provided. The *getClass* is the same as described in Algorithm 4. The *getOtherClass* function returns the already mapped class that corresponds to the entity other than the one provided in the relation provided. The *findNonAspect* returns a class of non aspect type in the collection provided. The *findResolvedAspect* returns a class of aspect type that has its existential dependency already set from the collection provided.

8.3.7 Association-like relations relations mapping

Most of the relation types are mapped using associations. This means they are copied as is to the OntoObjectModel to retain as much of the OntoUML semantics as possible to allow for their implementation in the target language to be adapted in the later stages. The relation types mapped this way are: Material, Mediation, Association, Characterization and ComponentOf.

8.3.8 SubQuantityOf, SubCollectionOf and MemberOf relations mapping

SubQuantityOf, SubCollectionOf and MemberOf relations are handled in the same way that is different form other relations (see section 8.3.7 for those). To a certain extent, this mapping is similar to the mapping of Phases (see section 8.3.4), the algorithm is listed in Algorithm 7.

Algorithm 6 Determinig existential dependencies

```
1:  procedure processExistentialDependencies(model)
2:      queue ← new Queue()
3:      for all entity ∈ model that is an aspect type do
4:          characterizations ← getCharacterizations(entity, model)
5:          entityClass ← getClass(entity)
6:          related ← []
7:          for all ch ∈ characterizations do
8:              related.push(getOtherClass(ch, entity))
9:          queue.enqueue({class ← entityClass, related ← related})
10:     limit ← (queue.length)^2
11:     while !queue.empty ∧ limit > 0 do
12:         current ← queue.front; queue.dequeue()
13:         dependency ← findNonAspect(current.related)
14:         if dependency is not null then
15:             setDependecy(currentClass, dependency)
16:         else
17:             dependency ← findResolvedAspect(current.related)
18:             if dependency is not null then
19:                 setDependecy(currentClass, dependency)
20:             else
21:                 queue.enqueue(current)
22:         limit ← limit − 1
23:     if queue.length > 0 then
24:         error("Invalid aspects in model.")
```

The cardinality has to be set to many to many as I could not come up with a way to handle situations when different members (and sub-quantities and sub-collections) have different cardinality of their relations to the same owner. To allow for all the variants, the cardinality is set as broad as possible.

The *getClass*, *addClass* and *addRelation* functions are the same as described in see section 8.3.4.

Algorithm 7 Mapping of SubQuantityOf, SubCollectionOf and MemberOf

```
1: procedure mapRelation(type, interfaceSuffix, attributeSuffix)
2:     groups ← group all relations of type type in model by their source end entity
3:     for all (relations, source) ∈ groups do
4:         if length of relations > 1 then                    ▷ add interface if needed
5:             targetTypeName ← source + interfaceSuffix
6:             addClass({
7:               name ← targetTypeName,
8:               isInterface ← ⊤,
9:             })
10:            for all relation ∈ relations do
11:                targetClass ← getClass(relation)
12:                targetClass.implementing.push(targetTypeName)
                                   ▷ declare subitems implement the interface
13:        else
14:            targetTypeName ← relations[0].targetEnd.type
15:        ownerClass ← getClass(source)
16:        addRelation({
17:            name ← partitionName,
18:            isInseparable ← ⊤, allowDuplicates ← ⊤,
19:            sourceEnd ← {
20:                className ← ownerClass.name,
21:                name ← ownerClass.name,
22:                maxItems ← −1, minItems ← 0
23:            },
24:            targetEnd ← {
25:                className ← targetTypeName,
26:                name ← targetTypeName,
27:                maxItems ← −1, minItems ← 0
28:            }
29:        })
```

Chapter 9

OntoObjectModel into target language view model transformation

This chapter covers the transformation from OntoObjectModel as a PIM into the target language's view model – a PSM. We will discuss the purpose and interface first and then the implementation for C# will be described.

9.1 Purpose and interface

As stated before, the purpose of this transformation is to transform OntoObjectModel into target language's view model which is then to be rendered by a renderer (see chapter 10). Its interface is simple:

```
interface ILanguageMapper {
  modelToViewModel: (
    model: OntoObjectModel,
    options: IOptions
  ) => Q.Promise<any>;
}
```

Listing 9.1: ILanguageMapper interface

The single method takes the OntoObjectModel and application options (see section 6.3.3) and returns a Promise to the resulting view model.

9.2 Primitive type mapping

For typed languages, `ILanguageMapper` can also perform mapping from domain types used in OntoObjectModel (e.g. `money`) to platform specific types (e.g. `decimal` in C#). This is made possible by the `typeMapping` application

parameter (see section 6.3.3). If specified, it should point to a JSON file with the mapping. Its structure is simple, just key–value pairs of domain type–platform type. This file is validated using a JSON Schema. If a type is not found in the mapping file, it is copied as is to the result.

9.3 C# view model representation

This section covers the C# view model that is used in rendering (see section 10.2). It extends OntoObjectModel with various platform specific attributes needed in rendering. Each of the added attribute will be linked with the corresponding rendering part to illustrate its purpose. Some of the properties introduced could be inferred from other properties, however the templating engine may not support complex conditions so these 'redundant' properties are added to facilitate the rendering.

9.3.1 ModelViewModel

The whole model is represented by an instance of `ModelViewModel`. It has two properties:

- `classes` – all the classes in the model, also covering all the relations among them (`ClassViewModel` instnaces, see section 9.3.2)
- `namespace` – name of the namespace the generated classes should be in (see section 4.4)

As opposed to OntoObjectModel, relations are described inside classes, not on the root level (see section 8.2.7).

9.3.2 ClassViewModel

Every class is represented by instances of `ClassViewModel`. It has the following properties:

- `name`, `isInterface`, `existentiallyDependentOn` – these are analogous to their `ClassInfo` counterparts (see section 8.2.2)
- `superClass`, `unionClasses`, `implementing` – also analogous to their `ClassInfo` counterparts, only difference being they contain reference to the corresponding view model(s), not the name
- `isOverlapping` – flag indicating the class originated as a result of overlapping inheritance (see section 8.3.3 for definition and section 10.2.9 for usage)

- `props` – all the public properties of the class (`PropertyViewModel` instances, see section 9.3.3), for usage, see section 10.2.5
- `methods` – all the public methods of the class (`MethodViewModel` instances, see section 9.3.5), for usage, see section 10.2.4
- `ctor` – constructor information (instance of `CtorViewModel`, see section 9.3.7), this is used for example while handling essential relations, see section 10.2.7
- `interfaceExtends` – names of the interfaces the class' interface extends (see section 10.2.2)
- `classExtends` – names of the classes or interfaces the class extends (see section 10.2.2)
- `relations` – information about all the relations this class is a part of (`RelationViewModel` instances, see section 9.3.8)
- `derivedRelations` – information about all the derived relations this class is a part of (`DerivedRelationViewModel` instances, see section 9.3.9), see section 10.2.8 for usage

Note that the `isAbstract` flag is missing. The reason for this is that due to the way multiple inheritance is implemented (via composition), it is sometimes necessary to instantiate classes marked as abstract, so this flag is ignored altogether (see section 10.2.2 for details).

9.3.3 PropertyViewModel

In C# attributes are modelled using properties (see section 4.2), and so every attribute of a class is represented by a `PropertyViewModel` instance. Its properties are:

- `name` – name of the property
- `minItems` – minimal count of items in the property
- `maxItems` – maximal count of items in the property (`null` for unlimited)
- `typeInfo` – information about the type of the property (instance of `TypeInfoViewModel`, see section 9.3.4)
- `isCollection` – flag indicating the property represents a collection of items
- `hasConstraints` – flag indicating the has at least one constraint on the count of the items it stores

9.3.4 TypeInfoViewModel

Instances of `TypeInfoViewModel` store information about types of properties or method arguments. It is very similar to `TypeInfo` (see section 8.2.4), it only adds two properties:

- `isInterface` – flag indicating the type should reference its type by its interface (see section 10.2.2)
- `shouldMakeNullable` – flag indicating that the type represents a value type and should be made nullable (see section 4.1 for definition of nullable and section 9.4.2 for usage)

9.3.5 MethodViewModel

There is nothing to add when describing methods in C#, the `MethodViewModel` is therefore very close to `MethodInfo` (see section 8.2.5), the only difference being the reference properties refer to view model types (`TypeInfoViewModel` and `ParameterViewModel`, see section 9.3.6).

9.3.6 ParameterViewModel

As with `MethodViewModel`, `ParameterViewModel` is similar to `ParameterInfo`, see section 8.2.6, the `typeInfo` understandably references to an instance of `TypeInfoViewModel`.

9.3.7 CtorViewModel

Constructors are described using instances of `CtorViewModel`. It has these properties:

- `parameters` – every parameter of the constructor (`ParameterViewModel` instances, see section 9.3.6)
- `relations` – mapping between parameter name and the relation it initializes, see section 10.2.7
- `parentParameterNames` – names of the parameters (included in the `parameters` property), that should be used to initialize the parent class (see section 4.3 for details and section 10.2.7 for usage)

9.3.8 RelationViewModel

`RelationViewModel` is used to represent a relation a given class is a member of. All the properties are listed below:

- `name`, `allowDuplicates` – analogous to their `RelationInfo` counterparts (see section 8.2.7)
- `type` – information about the type of the relation (1:1, 1:N or M:N)
- `isSource` – flag indicating the given class is at the source end of the relation (e.g. the '1' in 1:N)
- `sourceClassName`, `targetClassName` – names of the source and target classes respectively
- `hasSet` – flag indicating the given class can add members of the relation
- `hasUnset` – flag indicating the given class can remove members of the relation
- `minItems` – minimal count of items in the relation
- `maxItems` – maximal count of items in the relation (`null` for unlimited)
- `hasConstraints` – flag indicating the has at least one constraint on the count of the items that can be in this relation end
- `shouldRenderField` – flag indicating the given class should render the field backing the relation, see section 10.2.6
- `shouldInvalidateOnRemove` – flag indicating the given class should invalidate the items when they are unset or removed, see section 10.2.10

It also contains several 'redundant' properties to facilitate rendering:

- `isOneToOne`, `isOneToMany` and `isManyToMany` – denormalized information about relation type (see section 10.2.6)
- `otherClassName` – name of the class that is not the given class (i.e. `targetClassName` if `isSource` is true)

9.3.9 DerivedRelationViewModel

`DerivedRelationViewModel` instances provide all the properties needed to render a derived relation (see section 10.2.8):

- `name` – name of the derived relation
- `relatorName` – name of the relator the relation is derived from
- `otherClassName` – name of the class of the results of the derived relation
- `otherItemName` – name of the field in the class that points to the results of the derived relation

- `relatorOtherItemName` – name of the field in the relator that points to the resulting instances of this derived relation

There are also some 'redundant' properties to facilitate rendering:

- `isManyRelators` – flag indicating there can be more than one instance of the relator associated to the class
- `isManyOthers` – flag indicating there can be more than one instance of the other class associated to the relator
- `isManyResults` – flag indicating there can be more than one result of the relation

9.4 Implementation

This section describes the C# language mapper that was implemented to provide input to the C# renderer (see section 10.2). The notation used here is described in section 8.3.1.

9.4.1 Basic mapping

The overall order of the mapping is as follows:

1. Model for *ICanValidate* is added (see section 10.2.10).
2. Every class is mapped using the *mapClass* function (see section 9.4.5).
3. Every non-derived relation is mapped using the *mapRelation* function (see section 9.4.6)
4. Every derived relation is mapped using the *mapDerivedRelation* function (see section 9.4.7)
5. All the classes' constructors are updated using the *updateCtorParameters* function (see section 9.4.8)

All the mapping algorithms are described in the following sections. The order is chosen so that the mappings used in other mappings are described first.

Algorithm 8 Mapping of TypeInfo to TypeInfoViewModel

```
1:  function mapTypeInfo(typeInfo, checkNullable)
2:      name ← tryMapPrimitiveType(typeInfo.name)
3:      result ← {
4:          name ← name,
5:          isInterface ← ⊥, isReference ← typeInfo.isReference,
6:          shouldMakeNullable ← checkNullable ∧ isValueType(name)
7:      }
8:      if typeInfo.isReference then
9:          ref ← getClassInfo(typeInfo.name)
10:         result.name ← ref.name
11:         result.isInterface ← ref.isInterface
12:         result.shouldMakeNullable ← ⊥
13:     return result
```

9.4.2 TypeInfo mapping

As the `TypeInfoViewModel` is very similar to `TypeInfo` (see section 9.3.4) the mapping is straight forward, it is listed in algorithm 8.

The *tryMapPrimitiveType* performs the primitive type mapping (see section 9.2), the *getClassInfo* function returns a `ClassInfo` instance with the name specified. The function *isValueType* returns true if the type name provided to it represents a C# value type and therefore should be made nullable (see section 4.1) to be able to distinguish for example between 0 and *not set* values for `int` property. These types are considered to be value types: `bool`, `byte`, `char`, `decimal`, `double`, `float`, `int`, `long`, `sbyte`, `short`, `uint`, `ulong`, `ushort` (adapted from [66]).

9.4.3 Method mapping

Methods are mapped very simply by copying all the attributes from the `MethodInfo` instance, performing type mapping described in section 9.4.2 on the return type and all the parameters' types without checking for nullable types. There is no added logic or properties.

9.4.4 Property mapping

Mapping of properties from `AttributeInfo` instances is simple (algorithm 9). It copies the properties, performs primitive type mapping (*mapTypeInfo* function) as described in section 9.4.2 checking for nullable types and computes the auxilliary flags.

Algorithm 9 Mapping of AttributeInfo to PropertyViewModel

```
1: function getMaxItems(n)
2:     if n > 0 then
3:         return n
4:     else
5:         return null
6: function mapAttribute(attr)
7:     return {
8:         name ← attr.name,
9:         typeInfo ← mapTypeInfo(attr.type),
10:        minItems ← attr.minItems,
11:        maxItems ← getMaxItems(attr.maxItems),
12:        isCollection ← attr.minItems > 1 ∨ attr.maxItems > 1
13:                ∨attr.maxItems < 0,
14:        hasConstraints ← attr.minItems > 1 ∨ attr.maxItems > 1
15:    }
```

9.4.5 Class mapping

Classes are mapped using the recursive function described in algorithm 10. It memoizes the results to be effective.

Algorithm 10 Mapping of ClassInfo to ClassViewModel

```
1: function mapClass(c)
2:     if isMapped(c) then
3:         return getMappedClass(c)
4:     return {
5:         name ← c.name,
6:         isInterface ← c.isInterface,
7:         isOverlapping ← length of c.unionClasses > 0,
8:         existentiallyDependentOn ← c.existentiallyDependentOn
9:         methods ← mapMethods(c)
10:        props ← mapAttributes(c)
11:        superClass ← mapClass(c.superClass)
12:        unionClasses ← mapClasses(c.unionClasses)
13:        implementing ← mapClasses(c.implementing)
14:        classExtends ← getClassExtends(c)
15:        interfaceExtends ← getInterfaceExtends(c)
16:    }
```

The *isMapped* function determines if the argument passed to it has already been mapped and *getMappedClass* returns the result of that mapping (these perform the memoization). The *mapMethods* function maps *mapMethod* from

section 9.4.3 on the array provided and similarly *mapAttributes* maps the *mapAttribute* from section 9.4.3 and so does *mapClasses* with the *mapClass* described here.

The *getClassExtends* function returns an array of names of the items the mapped class should extend:

- own interface (see section 10.2.2)
- the super class

Analogously, the *getInterfaceExtends* function returns an array of names of items the mapped class' interface should extend (see section 10.2.2 for why there are both class and interface):

- the `ICanValidate` interface (see section 10.2.10)
- the super class interface (unless the class is overlapping)
- all the items from `unionClasses`
- all the items from `implementing`

9.4.6 Relation mapping

Non-derived relations are iterated over and added to the classes they are related to (for derived relations mapping see section 9.4.6). The procedure is listed in algorithm 11. It calculates various flags that apply to each end of the relation and are used when rendering. Also if the relation is essential, the constructor of the source class (i.e. the whole class) is updated to ensure the essential part is initialized upon creation. The reason source and target ends are treated differently is mainly caused by the fact that in essential and inseparable relations, the whole is always the source.

The *getRelationType* function returns the relation type (1:1, 1:N or M:N). The *getMappedClass* function is the one described in algorithm 10, *mapTypeInfo* is defined in algorithm 8 and *getMaxItems* is the same as in algorithm 9.

9.4.7 Derived relation mapping

Derived relations must be mapped after the regular ones (see section 9.4.6) as the mapping uses the already mapped relations. The procedures needed are listed in algorithm 12. They basically copy the correct attributes to the properties needed by rendering (see section 10.2.8).

The *mapDerivedRelationEnd* function returns a mapped end with computed auxiliary flags. It is then used by the *mapDerivedRelation* procedure that uses it to map both source ans target ends of the relation. The *getRelationFrom* returns a relation that connects the relator and the class provided. The *getMappedClass* function is the one described in algorithm 10.

Algorithm 11 Mapping of RelationInfo to ClassViewModels

Require: In case of 1:N relation, the source end is poniting to the '1' end

procedure mapRelation(r)
  $relationType \leftarrow getRelationType(r)$
  $sourceClass \leftarrow getMappedClass(r.sourceEnd.className)$
  $targetClass \leftarrow getMappedClass(r.targetEnd.className)$
  $isEssentialOrInseparable \leftarrow r.isEssential$ or $r.isInseparable$
  $isSourceAspect \leftarrow sourceClass.existentiallyDependentOn$ is not null
  $isTargetAspect \leftarrow targetClass.existentiallyDependentOn$ is not null
  $hasSourceSet \leftarrow (\neg r.isEssential)$
    $\wedge\neg(isSourceAspect \wedge \neg isTargetAspect)$
  $hasTargetSet \leftarrow \neg isEssentialOrInseparable \wedge \neg isTargetAspect$
  $sourceRelation \leftarrow \{$
    $name \leftarrow r.name,$
    $type \leftarrow relationType,$
    $isOneToOne \leftarrow relationType = ONE\_TO\_ONE,$
    $isOneToMany \leftarrow relationType = ONE\_TO\_MANY,$
    $isManyToMany \leftarrow relationType = MANY\_TO\_MANY,$
    $allowDuplicates \leftarrow r.allowDuplicates,$
    $isSource \leftarrow \top,$
    $sourceClassName \leftarrow sourceClass.name$
    $targetClassName \leftarrow targetClass.name$
    $otherClassName \leftarrow targetClass.name$
    $otherItemName \leftarrow r.targetEnd.name$ or $targetClass.name$
    $hasSet \leftarrow hasSourceSet$
    $hasUnset \leftarrow hasSourceSet$
      $\wedge\neg(r.targetEnd.minItems = r.targetEnd.maxItems = 1)$
    $minItems \leftarrow r.targetEnd.minItems$
    $maxItems \leftarrow getMaxItems(r.targetEnd.maxItems)$
    $shouldRenderField \leftarrow \top,$
    $shouldInvalidateOnRemove \leftarrow$
      $targetClass.existentiallyDependentOn = sourceClass.name$
  $\}$
  $sourceClass.relations.push(sourceRelation)$ ▷ add to source class

```
33:     targetRelation ← {
34:         name ← r.name,
35:         type ← relationType,
36:         isOneToOne ← relationType = ONE_TO_ONE,
37:         isOneToMany ← relationType = ONE_TO_MANY,
38:         isManyToMany ← relationType = MANY_TO_MANY,
39:         allowDuplicates ← r.allowDuplicates,
40:         isSource ← ⊥,
41:         sourceClassName ← sourceClass.name
42:         targetClassName ← targetClass.name
43:         otherClassName ← sourceClass.name
44:         otherItemName ← r.sourceEnd.name or sourceClass.name
45:         hasSet ← hasTargetSet
46:         hasUnset ← hasTargetSet
47:             ∧¬(r.sourceEnd.minItems = r.sourceEnd.maxItems = 1)
48:         minItems ← r.sourceEnd.minItems
49:         maxItems ← getMaxItems(r.sourceEnd.maxItems)
50:         shouldRenderField ← sourceClass.name ≠ targetClass.name,
51:         shouldInvalidateOnRemove ←
52:             sourceClass.existentiallyDependentOn = targetClass.name
53:     }
54:     targetClass.relations.push(targetRelation)        ▷ add to target class
55:     if r.isEssential ∨ r.isPartInitializedWithWhole then
56:         sourceClass.ctor.parameters.push({
57:             name ← r.targetEnd.name
58:             typeInfo ← mapTypeInfo({
59:                 name ← targetClass.name, isReference ← ⊤
60:             })
61:             isCollection ← r.targetEnd.maxItems > 1
62:                 ∨r.targetEnd.maxItems < 0
63:         })
64:         sourceClass.ctor.relations.push({
65:             parameterName ← r.targetEnd.name
66:             relation ← targetRelation
67:         })
                    ▷ add ctor parameter and relation for essential relations
```

Algorithm 12 Mapping of DerivedRelationViewModel

```
1: function mapDerivedRelationEnd(r, rToThis, rToOther, otherEnd,
   relator)
2:     result ← {
3:         name ← r.name,
4:         relatorName ← relator.isInterface,
5:         otherClassName ← otherEnd.className,
6:         otherItemName ← otherEnd.name,
7:         relatorOtherItemName ← relationToOther.otherItemName,
8:         isManyRelators ← rToThis.isManyToMany
9:             ∨(rToThis.isOneToMany ∧ ¬rToThis.isSource)
10:        isManyOthers ← rToOther.isManyToMany
11:            ∨(rToOther.isOneToMany ∧ rToOther.isSource)
12:    }
13:    result.isManyResults ← result.isManyRelators
14:        ∨result.isManyOthers
15:    return result
16: procedure mapDerivedRelation(r)
17:    relator ← getMappedClass(r.derivedFrom)
18:    rToSource ← getRelationFrom(relator, r.sourceEnd.className)
19:    rToTarget ← getRelationFrom(relator, r.targetEnd.className)
20:    source ← getMappedClass(r.sourceEnd.className)
21:    target ← getMappedClass(r.targetEnd.className)
22:    source.derivedRelations.push(
23:        mapDerivedRelationEnd(r, rToSource, rToTarget,
24:            r.TargetEnd, relator))
25:    target.derivedRelations.push(
26:        mapDerivedRelationEnd(r, rToTarget, rToSource,
27:            r.SourceEnd, relator))
```

9.4.8 Constructor mapping

The constructor parameters can be modified from various places – e.g. when mapping an essential relation (see algorithm 11). The procedure listed in algorithm 13 ensures that classes derived from classes with constructor parameters have those constructor parameters added as well. It iterates over the superClass' constructor parameters and adds it to the current class constructor parameters and similarly for each of the union classes.

Algorithm 13 Updating the constructor parameters

```
1: procedure updateCtorParameters(c)
2:     for all p ∈ c.superClass.ctor.parameters do
3:         c.ctor.parentParameterNames.push(p.name)
4:         c.ctor.parameters.pushUnique(p)
5:     for all uc ∈ c.unionClasses do
6:         for all p ∈ uc.ctor.parameters do
7:             c.ctor.parameters.pushUnique(p)
```

The *pushUnique* method pushes an item to an array unless it is already contained in the array.

Chapter 10

Target language view model rendering

In this chapter, the process of rendering the target language source code is described. First the interface and purpose of the renderers is explained and then the C# renderer is introduced in detail.

10.1 Purpose and interface

Renderer is responsible for the transformation of a view model into the target language source code file/files. It may be implemented in various ways as long as it implements this interface:

```
interface ILanguageRenderer {
  generateCode: (model: any, options: IOptions)
    => Q.Promise<any>;
}
```

Listing 10.1: ILanguageRenderer interface

The interface consists of a single method that takes a view model (that can be of any type) and in instance of application options (see section 6.3.3) and returns a Promise to all the operations it needs to perform (typically writing to disk).

10.2 Implementation

This section describes the C# language renderer that was implemented (using the `handlebars` template engine, see section 6.2.2) to illustrate the viability of the solution presented in this thesis. The handling of various concerns will be covered in the following text.

10.2.1 Note on template notation

Rendering templates will be listed in the following form:

- variable.property – the value of *property* will be output
- `monospace text` – this will be output as is
- *italics text* – this denotes a call to another template the result of which will be output
- ⟨text in angle brackets⟩^{x} – this will be output if the condition x is satisfied
- ⟨text in angle brackets⟩^{*1} – this will be output if the condition explained under the listing using the specified number is satisfied
- ⟦(member of collection): content ⟧ – for every *member* of the *collection* the *content* will be output
- // text – this is a comment stating the template name and parameter

To see it in an example:

```
// templateName(parameterName: parameterType)
always⟨conditionally⟩^parameterName.property>2
  anotherTemplate(itsParameter) parameterName.property
```

Listing 10.2: Rendering template example

The content in the angle brackets will be rendered if the value specified by `parameterName.property` is greater than 2. There are also some utility string functions used throughout this chapter:

- *s.lowerCamel()* – renders s in CamelCase with the first letter in lower case
- *s.upperCamel()* – renders s in CamelCase with the first letter in upper case
- *s.upperSnake()* – renders s in UPPER_SNAKE_CASE
- *s.plural()* – renders s in plural form

Also, there are a few auxiliary templates:

```
// minItemsConst(x: {name: string;})
x.name.upperSnake()_MIN_ITEMS
// maxItemsConst(x: {name: string;})
x.name.upperSnake()_MAX_ITEMS
// associationField(r: RelationVeiwModel)
r.name.lowerCamel()Association
```

```
// toValidateField((x: {name: string;}))
x.name.lowerCamel()ToValidate
// countField((x: {name: string;}))
x.name.lowerCamel()Count
```

Listing 10.3: Render helpers

10.2.2 Basic rendering

Each entity in the model is rendered as an interface — class pair, unless the `isInterface` flag is set, then the class part is omitted (see section 8.2). The reason is that, as stated in section 4.3, C# does not allow for multiple inheritance, but we need to maintain the ability of an entity to inherit from multiple other entities. The solution for this is that instead of inheritance, the derived classes inherit only from its super class (see section 8.2.2) and implements the other predecessors' interfaces (and this is fine in C# to be done multiple times). Moreover, each class implements 'its own' interface.

The interface rendering template is listed in listing 10.4. It iterates over members and uses the corresponding templates to render their interfaces: properties (see section 10.2.5), methods (see section 10.2.4), relations (see section 10.2.6) and derived relations (see section 10.2.8).

```
// interface(c: ClassViewModel)
public interface Ic.name.upperCamel()⟨:⟩^{c.interfaceExtends is not empty}
  ⟦(n of c.interfaceExtends): In.upperCamel()⟨,⟩^{n is not last}⟧
{
  ⟦(p of c.props): propertyInterface(p)⟧
  ⟦(m of c.methods): methodSignature(m)⟧
  ⟦(r of c.relations):
    ⟨oneSideRelationInterface(r)⟩^{r.isOneToOne∨(r.isOneToMany∧¬r.isSource)}
    ⟨manySideRelationInterface(r)⟩^{r.isManyToMany∨(r.isOneToMany∧r.isSource)}
  ⟧
  ⟦(r of c.derivedRelations):
    ⟨oneResultDerivedInterface(r)⟩^{¬r.isManyResults}
    ⟨manyResultsDerivedInterface(r)⟩^{r.isManyResults}
  ⟧
}
```

Listing 10.4: Interface render

The template for rendering classes is listed in listing 10.5. Similarly to interfaces it merely calls various other templates to render different things: constructor (see section 10.2.7), relations (section 10.2.6) and derived relations (section 10.2.8), union classes inner instances and delegations (section 10.2.9), properties (section 10.2.5), methods (section 10.2.4) and utility interface implementations (section 10.2.10).

```
// class(c: ClassViewModel)
public class c.name.upperCamel()⟨:⟩^{c.classExtends is not empty}
⟦(n of c.classExtends):
  ⟨I⟩^{n is not superClass}n.upperCamel()⟨,⟩^{n is not last}⟧
{
  constructor(c)
  ⟦(r of c.relations): relationBody(r) ⟧
  ⟦(r of c.derivedRelations): derivedRelation(r)⟧
  ⟦(uc of c.unionClasses):
  private readonly Iuc.name.upperCamel()
    uc.name.lowerCamel();
  ⟧
  overlappingSuperClassProps(c)
  overlappingSuperClassMethods(c)
  unionClassesProps(c)
  unionClassesMethods(c)
  unionClassesRelations(c)
  unionClassesDerivedRelations(c)
  ⟦(p of c.props): propertyDeclaration(p)⟧
  ⟦(m of c.methods): methodDeclaration(m)⟧
  ⟦(i of c.implementing):
    ⟦(p of i.props): propertyDeclaration(p)⟧
    ⟦(m of i.methods): methodDeclaration(m)⟧
    ⟦(r of i.relations): relationBody(r)⟧
  ⟧
  invalidate(c)
  isValid(c)
}
```

Listing 10.5: Class render

Every file's content is rendered using the template listed in listing 10.6. It wraps the file content into a namespace declaration (see section 4.4) and adds references to the namespaces needed for the following reasons:

- `Ccmi.OntoUml.Utilities.AssociationClasses` – provides classes to implement associations (see section 6.4.1)
- `Ccmi.OntoUml.Utilities.Collections` – provides classes representing bounded collections (see section 6.4.2)
- `System` – contains definitions of exceptions used in the implementation
- `System.Collections.Generic` – the `IEnumerable<T>` interface is defined here

- `System.Linq` – contains definitions of `Select`, `SelectMany`, `Count` and `Distinct` methods

```
// file(content: string, namespaceName: string)
using Ccmi.OntoUml.Utilities.AssociationClasses;
using Ccmi.OntoUml.Utilities.Collections;
using System;
using System.Collections.Generic;
using System.Linq;

namespace
  ⟨namespaceName.upperCamel()⟩^namespace is not empty
  ⟨DefaultNamespace⟩^namespace is empty
{
  content
}
```

Listing 10.6: File render

10.2.3 Type rendering

Types are rendered in two steps. First the type name is determined:

```
// typeName(typeInfo: TypeInfoViewModel)
⟨I⟩*1 typeInfo.name⟨.upperCamel()⟩*1⟨?⟩*2
```

Listing 10.7: Type name render

The items in the numbered angle brackets are rendered if

1. `typeInfo.isInterface` or `typeInfo.isReference` is true
2. `typeInfo.shouldMakeNullable` is true

respectively (see section 9.3.4). Then type signatures for various usages are rendered for use in other templates:

```
// typeSignature(t: TypeInfoViewModel)
⟨IBoundedCollection<⟩^t.isCollection typeName(typeInfo)⟨>⟩^t.isCollection

// ctorTypeSignature(t: TypeInfoViewModel)
typeName(typeInfo)⟨[]⟩^t.isCollection
```

Listing 10.8: Type signature render

10.2.4 Method rendering

Own methods and methods from implemented interfaces are rendered in a very simple way (for rendering of methods delegated to the union classes instances see section 10.2.9):

```
// parameterList(m: MethodViewModel)
⟦(p of m.parameters):
  typeSignature(p.type) p.name.lowerCamel()⟨,⟩^{p is not last}
⟧

// methodSignature(m: MethodViewModel)
⟨typeSignature(m.typeInfo)⟩^{m.typeInfo is not null}⟨void⟩^{m.typeInfo is null}
  m.name.upperCamel()(parameterList(m))

// methodDeclaration(method: MethodViewModel)
public methodSignature(method)
{
  throw new NotImplementedException();
}

// methodCall(m: MethodViewModel)
m.name.upperCamel()(
  ⟦(p of m.parameters): p.name.lowerCamel()⟨,⟩^{p is not last}⟧
);
```

Listing 10.9: Method render

As we can see method signature is rendered using type rendering described in section 10.2.3 using `void` if no return type is provided. The method body is rendered to throw `NotImplementedException` to indicate that the actual body has to be written by hand. There is also a helper template to call a method defined that is used in other templates.

10.2.5 Property rendering

Properties are rendered in the way shown in listing 10.10. If a property has an upper bound on the count of items it can contain, this bound is enforced by the check in the property setter (for the flag reference see section 9.3.3) and so the collection assigned must be big enough. Also, properties are marked as `virtual` to allow for overriding them (this is used in section 10.2.9). Other than that, the rendering is pretty straight-forward.

```
// propertyInterface(prop: PropertyViewModel)
typeSignature(prop.typeInfo) prop.name.upperCamel() { get; set; }

// propertyBackingField(prop: PropertyViewModel)
```

```
private typeSignature(prop.typeInfo) prop.name.lowerCamel();

// setterContraints(prop: PropertyViewModel)
⟨if (value != null)
{
  ⟨const int MAX_ITEMS = prop.maxItems;
  if (value.MaxItems > MAX_ITEMS)
    throw new InvalidOperationException("The source collection
    for prop.name.upperCamel() is too small.");
  value.MaxItems = MAX_ITEMS;⟩prop.maxItems>1
}⟩prop.hasConstraints

// propertyDeclaration(prop: PropertyViewModel)
propertyBackingField(prop)
public virtual typeSignature(prop.typeInfo) prop.name
{
  get { return prop.name.toLower(); }
  set
  {
    setterContraints(prop)
    prop.name.lowerCamel() = value;
  }
}
```

Listing 10.10: Property render

10.2.6 Relation rendering

Rendering of relations uses the utility classes in the utilities library (described in section 6.4.1). The interface rendering methods listed in listing 10.12 just render appropriate calls to those classes creating relevant names for the wrapping methods and their parameters using the helpers from listing 10.11.

```
// relationGet(r: RelationViewModel)
Getr.otherItemName.lowerCamel()
// relationGetMany(r: RelationViewModel)
Getr.otherItemName.plural().lowerCamel()
// relationSet(r: RelationViewModel)
Setr.otherItemName.lowerCamel()
// relationUnset(r: RelationViewModel)
Unsetr.otherItemName.lowerCamel()
// relationAdd(r: RelationViewModel)
Addr.otherItemName.lowerCamel()
// relationRemove(r: RelationViewModel)
Remover.otherItemName.lowerCamel()
```

```
// relationMethodParameters(r: RelationViewModel)
⟨this, r.otherItemName.lowerCamel()⟩^r.isSource
⟨r.otherItemName.lowerCamel(), this⟩^¬r.isSource
```

Listing 10.11: Relation render helpers

```
// oneSideRelationInterface(r: RelationViewModel)
⟨void relationSet(r)
  (Ir.otherClassName.upperCamel()
    r.otherItemName.lowerCamel()
  );⟩^r.hasSet
⟨void relationUnset(r)();⟩^r.hasUnset
Ir.otherClassName.upperCamel() relationGet(r)();

// manySideRelationInterface(r: RelationViewModel)
⟨void relationAdd(r)
  (Ir.otherClassName.upperCamel()
    r.otherItemName.lowerCamel()
  );⟩^r.hasSet
⟨void relationRemove(r)
  (Ir.otherClassName.upperCamel()
    r.otherItemName.lowerCamel()
  );⟩^r.hasUnset
IEnumerable<Ir.otherClassName.upperCamel()>
  relationGetMany(r)();
```

Listing 10.12: Relation interface render

As we can see, rendering of setters and unsetters is managed by their respective flags. Getters are always rendered.

Listings 10.14, 10.15 and 10.16 show rendering of the relation bodies. First the backing field is rendered if the *shouldRenderBackingField* flag is set. This prevents situations when a class is in a relation with itself and the backing field would be rendered twice (see section 9.4.6). Then setters and unsetters are rendered using helpers defined in listing 10.13 and finally the getter is rendered wrapping the correct Get call of the appropriate association class.

```
// oneSideSetUnset(r: RelationViewModel)
⟨public void relationSet(r)(
  Ir.otherClassName.upperCamel()
    r.otherItemName.lowerCamel())
{
  relationUnset(r)();
  associationField(r).CreateLink(relationMethodParameters(r));
}⟩^r.hasSet
```

```
⟨private⟩^{r.hasSet∧¬r.hasUnset}⟨public⟩^{r.hasUnset}  relationUnset(r)()
{
   ⟨relationGet(r)()?.Invalidate();⟩^{r.shouldInvalidateOnRemove}
   associationField(r).DestroyLink(
      ⟨associationField(r).GetOne(this), this⟩^{r.isOneToMany}
      ⟨this, associationField(r).GetRight(this)⟩^{r.isOneToOne∧r.isSource}
      ⟨associationField(r).GetLeft(this), this⟩^{r.isOneToOne∧¬r.isSource}
      );
}

// manySideSetUnset(r: RelationViewModel)
⟨public void relationAdd(r)(
   Ir.otherClassName.upperCamel()
      r.otherItemName.lowerCamel())
=> associationField(r).CreateLink(relationMethodParameters(r));⟩^{r.hasSet}

⟨public relationUnset(r)()
{
   ⟨relationGet(r)()?.Invalidate();⟩^{r.shouldInvalidateOnRemove}
   associationField(r).DestroyLink(relationMethodParameters(r));
}⟩^{r.hasUnset}

// associationClass(r: RelationViewModel)
⟨OneToOne⟩^{r.isOneToOne}⟨OneToMany⟩^{r.isOneToMany}⟨ManyToMany⟩^{r.isManyToMany}
Association<
   Ir.sourceClassName.upperCamel(),
   Ir.targetClassName.upperCamel()>
```

Listing 10.13: Relation body render helpers

```
// oneToOneRelationBody(r: RelationViewModel)
⟨private static associationClass(r)
   associationField(r) = associationClass(r).Instance();
⟩^{r.shouldRenderField}
oneSideSetUnset(r)
public Ir.otherClassName.upperCamel()  relationGet(r)()
   => associationField(r).Get⟨Right⟩^{r.isSource}⟨Left⟩^{¬r.isSource}(this);
```

Listing 10.14: 1:1 relation body render

```
// oneToManyRelationBody(r: RelationViewModel)
⟨private static associationClass(r)  associationField(r)
    = associationClass(r).Instance(⟨true⟩^{r.allowDuplicates});
⟩^{r.shouldRenderField}
⟨manySideSetUnset(r)
```

```
public IEnumerable<Ir.otherClassName.upperCamel()>
  relationGetMany(r)() => associationField(r).GetMany(this);
⟩r.isSource
⟨oneSideSetUnset(r)
public Ir.otherClassName.upperCamel() relationGet(r)()
  => associationField(r).GetOne(this);
⟩¬r.isSource
```

Listing 10.15: 1:N relation body render

```
// manyToManyRelationBody(r: RelationViewModel)
⟨private static associationClass(r) associationField(r)
   = associationClass(r).Instance(⟨true⟩r.allowDuplicates);
⟨manySideSetUnset(r)
public IEnumerable<Ir.otherClassName.upperCamel()>
  relationGetMany(r)()
  => associationField(r).Get⟨Ms⟩r.isSource⟨Ns⟩¬r.isSource(this);
```

Listing 10.16: M:N relation body render

```
// relationBody(r: RelationViewModel)
⟨oneToOneRelationBody(r)⟩r.isOneToOne)
⟨oneToManyRelationBody(r)⟩r.isOneToMany)
⟨manyToManyRelationBody(r)⟩r.isManyToMany)
```

Listing 10.17: Relation body render

10.2.7 Essential associations rendering

Setting the value of essential associations upon instance creation is enforced in the constructor. Its template is shown in listing 10.18. First, relevant parameters are passed to the parent constructor via the `base` call (see section 4.3) and then all parameters are checked for nulls (throwing an exception on null). Next, all the union class inner instances are initialized and finally, the appropriate relations are created.

```
// constructor(c: ClassViewModel)
public c.name.upperCamel()(
  ⟦(p of c.ctor.parameters):
  ctorTypeSignature(p.typeInfo) p.name.lowerCamel()⟨,⟩p is not last
  ⟧)⟨: base(
    ⟦(pn of c.ctor.parentParameterNames):
    pn.lowerCamel()⟨,⟩pn is not last
    ⟧)⟩c.ctor.parentParameterNames is not empty
{
  ⟦(p of c.ctor.parameters):
```

```
  if (p.name.lowerCamel() == null)
    throw new ArgumentNullException(
      nameof(p.name.lowerCamel()));
  ⟧
  ⟦(uc of c.unionClasses):
  uc.name.lowerCamel() = new uc.name.upperCamel()(
    ⟦(p of uc.ctor.parameters):
    p.name.lowerCamel()⟨,⟩^p is not last
    ⟧);
  ⟧
  ⟦(r of c.ctor.relations):
  associationField(r).CreateLink(this,
    r.parameterName.lowerCamel());
  ⟧);
}
```

Listing 10.18: Constructor render

10.2.8 Derived relations rendering

As managing of creating and destroying the relators would be too complicated, only getters are rendered for derived relations. The template is shown in listing 10.19. Its sole purpose is to generate appropriate method call chains according to the cardinality of the relevant fields in the class and the relator. It utilizes C#features like `Select` and `SelectMany` (for explanation of the difference see [67]) and the null-propagating operator and expression-bodied members to make the code more concise (see section 4.5).

```
// oneResultDerivedInterface(r: DerivedRelationViewModel)
public Ir.otherClassName.upperCamel() relationGet(r)()
// manyResultsDerivedInterface(r: DerivedRelationViewModel)
public IEnumerable<Ir.otherClassName.upperCamel()>
  relationGetMany(r)()
// derivedRelation(r: DerivedRelationViewModel)
⟨manyResultsDerivedInterface(r) =>
  ⟨Getr.relatorName.plural().upperCamel()()
    ⟨.SelectMany(
      r => r.Getr.relatorOtherItemName.plural().upperCamel()
      ()).Distinct();⟩^r.isManyOthers
    ⟨.Select(
      r => r.Getr.relatorOtherItemName.upperCamel()
      ()).Distinct();⟩^¬r.isManyOthers
  ⟩^r.isManyRelators
  ⟨Getr.relatorName.upperCamel()()
   ?.Getr.relatorOtherItemName.plural().upperCamel()()
```

```
        .Distinct();
   ⟩^¬r.isManyRelators
⟩^r.isManyResults
⟨oneResultDerivedInterface(r)
   => Getr.relatorName.upperCamel()()
     ?.Getr.relatorOtherItemName.upperCamel()();
⟩^¬r.isManyResults
```

Listing 10.19: Derived relation body render

10.2.9 Overlapping inheritance rendering

Overlapping inheritance is rendered using composition. This means the every class that is a result of overlapping inheritance (see section 8.3.3) contains a private instance of every union class it is composed of and it delegates the calls to their properties and methods to them. The `isOverlapping` is used to mark such classes (see section 9.3.2).

Every property of the super class must be handled in a way that on each set the properties of the inner union class instances are set as well as the property of the given overlapping class to keep the inner union class instances in sync. The same goes for methods – we need to call the method also on the union classes' instances in case the method has side effects. If the method should return a value, the result of the superclass version is returned. This handling is shown in listing 10.20, notice the use of the `override` keyword to override the inherited property or method (see section 4.3).

```
// overlappingSuperClassProps(c: ClassViewModel)
⟨⟦(p of c.superClass.props):
propertyBackingField(p)
public override typeSignature(p.typeInfo)p.name.upperCamel()
{
  get { return p.name.lowerCamel(); }
  set
  {
    setterContraints(p)
    p.name.lowerCamel() = value;
    ⟦(uc of c.unionClasses):
      uc.name.lowerCamel().p.name.upperCamel() = value;
    ⟧
  }
}⟧⟩^c.isOverlapping
// overlappingSuperClassMethods(c: ClassViewModel)
⟨⟦(m of c.superClass.methods):
public override methodSignature(m)
{
```

```
  ⟦(uc of c.unionClasses):
    uc.name.lowerCamel().methodCall(m)
  ⟧
  ⟨return ⟩^{m.typeInfo is not null} methodCall(m)
}⟧⟩^{c.isOverlapping}
```

Listing 10.20: Superclass prop decomposition

Each of the properties and methods that are to be realized by delegation to the inner union class instance need to be handled appropriately (see listing 10.21). They are simply delegated to the inner instance as is, transparently to the user of the overlapping class.

```
// unionClassesProps(c: ClassViewModel)
⟦(uc of c.unionClasses):
  ⟦(p of uc.props):
public typeSignature(p.typeInfo)p.name.upperCamel()
{
  get { return uc.name.lowerCamel().p.name.upperCamel(); }
  set { uc.name.lowerCamel().p.name.upperCamel() = value; }
} ⟧
⟧

// unionClassesMethods(c: ClassViewModel)
⟦(uc of c.unionClasses):
  ⟦(m of uc.methods):
public methodSignature(m) => uc.name.lowerCamel().methodCall(m)
  ⟧
⟧
```

Listing 10.21: Union classes property and method delegation

Similarly to properties and methods, relations are delegated as well (see listing 10.22). The main idea behind the delegation remains the same, only more methods need to be processed per relation.

```
// oneSideDelegation(r: RelationViewModel,
//    c: ClassViewModel)
⟨public void relationSet(r)(
  Ir.otherClassName.upperCamel()
    r.otherItemName.lowerCamel()
  ) => c.name.lowerCamel().relationSet(r)(
    r.otherItemName.lowerCamel());⟩^{r.hasSet}
⟨public void relationUnset(r)() =>
  c.name.lowerCamel().relationUnset(r)();⟩^{r.hasUnset}
public Ir.otherClassName.upperCamel() relationGet(r)() =>
  c.name.lowerCamel().relationGet(r)();
```

```
// manySideDelegation(r: RelationViewModel,
//     c: ClassViewModel)
⟨public void relationAdd(r)(
  Ir.otherClassName.upperCamel()
    r.otherItemName.lowerCamel()
  ) => c.name.lowerCamel().relationAdd(r)(
    r.otherItemName.lowerCamel());⟩^{r.hasSet}
⟨public void relationRemove(r)(
  Ir.otherClassName.upperCamel()
    r.otherItemName.lowerCamel()
  ) => c.name.lowerCamel().relationRemove(r)(
    r.otherItemName.lowerCamel());⟩^{r.hasUnset}
public IEnumerable<Ir.otherClassName.upperCamel()>
  relationGetMany(r)()
    => c.name.lowerCamel().relationGetMany(r)();

// unionClassesRelations(c: ClassViewModel)
⟦(uc of c.unionClasses):
  ⟦(r of uc.relations):
    ⟨oneSideDelegation(r, uc)⟩^{r.isOneToOne∨(r.isOneToMany∧¬r.isSource)}
    ⟨manySideDelegation(r, uc)⟩^{r.isManyToMany∨(r.isOneToMany∧r.isSource)}
  ⟧
⟧
```

Listing 10.22: Union classes relation delegation

Lastly, derived relations are delegated as well (see listing 10.23). The logic is still the same, just call the corresponding method of the inner union class instance.

```
// unionClassesDerivedRelations(c: ClassViewModel)
⟦(uc of c.unionClasses):
  ⟦(r of uc.derivedRelations):
⟨oneResultDerivedInterface(r)
  => uc.name.lowerCamel().relationGet(r)();
⟩^{¬r.isManyResults}
⟨manyResultsDerivedInterface(r)
  => uc.name.lowerCamel().relationGetMany(r)();
⟩^{r.isManyResults}
  ⟧
⟧
```

Listing 10.23: Union classes derived relation delegation

10.2.10 Soft checking

Constraints that are imposed on the model by some constructs that cannot be effectively enforced by other means in C# are dealt with using so called soft checking. This means that every class in the model provides a method that returns `true` if the instance is valid according to those constraints. This is achieved by the `ICanValidate` interface. It is listed in listing 10.24:

```
public interface ICanValidate
{
  bool IsValid(bool deep);
  void Invalidate();
}
```

Listing 10.24: ICanValidate

Every class' interface extends the `ICanValidate`, so on every class the `IsValid` and `Invalidate` methods can be called. The `IsValid` method returns a value indicating whether the instance does not violate any validity rules. If the *deep* parameter is set to true, IsValid is called also on all of the instances associated by relations.

After the `Invalidate` method is called, the instance should return false to all subsequent `IsValid` calls. Also once invalidated, the instance cannot be 're-validated'. This is used for indicating for example that an aspect was separated from its bearer (there is no easy way to destroy an instance deterministically in C#, so we must use this method to at least enable the soft checking).

The template of `Invalidate` is shown in listing 10.25. First a private flag field is rendered and then the method itself. The method is marked `virtual` in base classes to allow for the derived classes to `override` it according to their needs (see section 4.3). In it, the flag is raised and then every aspect dependent on the current class is removed from it and therefore invalidated (as we saw in section 10.2.6, aspects are invalidated upon removing from relation). This ensures that even the instances transitively dependent on the current class are properly invalidated recursively.

The template of `IsValid` can be seen in listing 10.26. The method is marked `virtual/override` as well. Inside the method the `isInvalidated` flag is checked, then all the inner instances of union classes are checked (see section 10.2.9) and finally constraints on relations – and optionally also their members – (section 10.2.6) and properties (section 10.2.5) are checked. If any of the checks fails, `false` is returned immediately, otherwise `true` is returned to indicate the instance is valid. The recursive validation utilizes the *Any* method from the `System.Linq` namespace (see [68]).

```
// removeAll(r: RelationViewModel)
foreach (var item in relationGetMany(r))
{
  relationRemove(r)(item);
```

```
}

// invalidate(c: ClassViewModel)
private bool isInvalidated = false;
public ⟨override⟩^{c.superClass}⟨virtual⟩^{¬c.superClass}void Invalidate()
{
  isInvalidated = true;
  [[(r of c.realtions)
  ⟨relationUnset(r)();⟩^{r.isOneToOne∨(r.isOneToMany∧¬r.isSource)}
  ⟨removeAll(r)⟩^{r.isManyToMany∨(r.isOneToMany∧r.isSource)}
  ]]
}
```

Listing 10.25: Invalidate render

```
// isValidOneEnd(r: RelationViewModel)
var toValidateField(r) = relationGet(r)();
if (associationField(r) == null
  ⟨|| toValidateField(r) == null⟩^{r.minItems>0}
  || (deep && !toValidateField(r).IsValid(deep))
  ) return false;

// isValidManyEnd(r: RelationViewModel)
⟨const int minItemsConst(r) = r.minItems;⟩^{r.minItems>0}
⟨const int maxItemsConst(r) = r.maxItems;⟩^{r.maxItems>1}
var toValidateField(r) = relationGetMany(r)();
⟨var countField(r) = toValidateField(r).Count();⟩^{r.hasConstraints}
  if (associationField(r) == null
    ⟨|| countField(r) < minItemsConst(r)⟩^{r.minItems>0}
    ⟨|| countField(r) > maxItemsConst(r)⟩^{r.maxItems>1}
    || (deep && toValidateField(r).Any(x => !x.IsValid(deep)))
    ) return false;

// isValid(c: ClassViewModel)
public ⟨override⟩^{c.superClass}⟨virtual⟩^{¬c.superClass}bool IsValid()
{
  ⟨if(isInvalidated) return false;⟩^{c.existentiallyDependentOn}
  [[(uc of c.unionClasses):
  if (uc.name.lowerCamel() == null
    || !uc.name.lowerCamel().IsValid()) return false;
  ]]
  [[(r of c.relations):
    ⟨isValidOneEnd(r)⟩^{r.isOneToOne∨(r.isOneToMany∧¬r.isSource)}
    ⟨isValidManyEnd(r)⟩^{r.isManyToMany∨(r.isOneToMany∧r.isSource)}
```

```
  ⟧
  ⟦(p of c.props):
    ⟨⟨const int minItemsConst(p) =p.minItems;⟩^{p.isCollection}
      if (p.name.upperCamel() == null
      ⟨|| !p.name.upperCamel()().Count()
        < minItemsConst(r)⟩^{p.isCollection}) return false;
    ⟩^{p.minItems>0}
  ⟧
  return true;
}
```

Listing 10.26: IsValid render

Part III

Assessment

Chapter 11

Case studies

To illustrate the code generation capabilities two sample models were created to illustrate the features implemented. The models were constructed to be as small as possible while showing the most of the features to keep the output focused. We will not discuss their domain usage or the problems they should solve, we will settle with that they are valid OntoUML models that make logical sense.

11.1 Case study 1

The diagram of the first sample model is shown in fig. 11.1. There is a *Vehicle* Kind with two overlapping Subkinds – *Plane* and *Boat* (the overlapping class can represent a seaplane for example). Every Boat can be registered at a *Port* playing the Role of a *Registrar* and every Port has a *Location*.

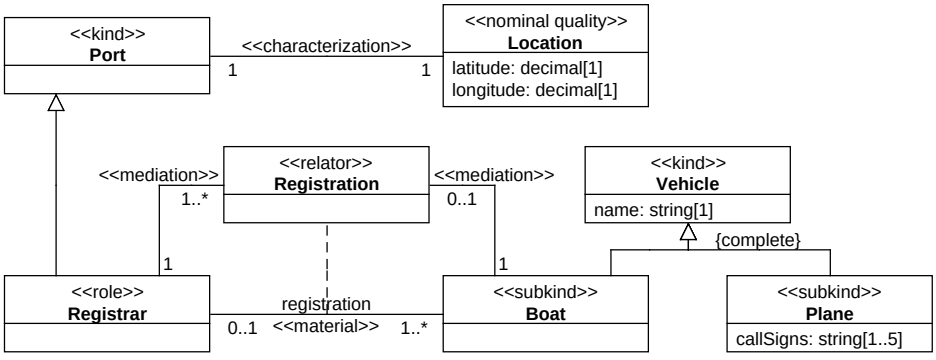


Figure 11.1: Case study 1

The code generated from this case study model is listed in Appendix B (the code has been manually reformatted to fit on the page and to conserve space the using statements and namespace declarations were omitted as they are the same across all the files). This model shows the following features:

- Relations
- Derived Relations
- Roles
- Attributes (both scalar and collections)
- Overlapping inheritance
- Aspects
- Soft checking

In the following sections the enumerated features implementation will be assessed.

11.1.1 Relations

The relations implementation can be seen in almost all the classes, a good example is the Registration class (that originated from a Relator). The code generated for it can be seen in listing B.8. We can see from its interface that it can Get a Boat and Set a Boat. It cannot Unset it however. This is because as seen in the model the cardinality for the boat is `1` meaning a Registration always has to have exactly one Boat associated with it – the Unset method is therefore not rendered as public to avoid breaking the modelled rules. It is implemented as private however, because it is needed internally while Setting (and thus replacing) the Boat.

Another example is the Registrar (listing B.7). Since it can have multiple Registrations attached, all three methods are public – *AddRegistration*, *RemoveRegistration* and *GetRegistrations*. However, there is a lower bound on the number of Registrations and this is checked in the *IsValid* method as checking the number on each removal would be potentially very inefficient.

11.1.2 Derived relations

There is a derived relation in the model between Boat and Registrar. Taking the Boat (listing B.1) as the illustration of the case with at most one result, we see that it has a method called *GetRegistrar*. In its implementation we see that the Registration is retrieved and from it the Registrar is returned.

Registrar (listing B.7) on the other hand can be associated with many boats and so its *GetBoats* method returns an `IEnumerable<Boat>` (see [63]).

The implementation correctly retrieves all the Registrations and, as there is at most one boat per Registration, calls the *Select* method with the correct getter lambda.

11.1.3 Roles

The Port's Registrar Role implementation can be seen in listing B.7. It is a standard class, that is connected to its owner as also seen in listing B.6. It does not inherit from the Port, as that would lead to a cyclic reference and also bring synchronization problems with the properties of both the Role and the owner. Instead it is connected using the Association class in a way that prohibits Registrar from detaching itself from the Port or change which one it is attached to. The Port however can let go of its Registrar roles invalidating them in the process. This goes very well with the semantics of Roles (see section 2.1.2).

11.1.4 Attributes

There are examples of both scalar attributes and collections in the model. For a scalar attribute example, take Vehicle (listing B.9) and its *name* attribute. It has been transformed to a property *Name* and since it is marked as required in the model (the cardinality is `1`), the fact that property is not null is checked in the *IsValid* method implementation.

Plane's *callSigns* attribute represents a collection example (listing B.5). It has both upper and lower bounds. The upper bound as it can be set even on an empty collection is enforced on the resulting property setter. The lower bound however is not enforced there to make the usage of the class more convenient (we want to be able to create the Plane with the collection empty and then add members to it). Instead the lower bound if checked in the *IsValid* method.

11.1.5 Overlapping inheritance

As the generalization set with the two Vehicle subtypes is overlapping, a new class called BoatPlane was generated (see section 10.2.9). The resulting C# code can bee seen in listing B.2. There we can see that its interface is correctly composed of the interfaces of both atomic classes. Inner private instances of the atomic classes are declared and initialized in the constructor. The *Name* property is overridden and kept in sync with the inner instances. The *CallSigns* property is delegated to the inner Plane instance and the relation methods are delegated to the Boat instance (both the relation to the Registration relator and the derived relation). In the *IsValid* method, both inner instances are validated (see section 10.2.10).

11.1.6 Aspects

In the model there is one entity of an aspect type – the Location Nominal Quality (listing B.4). As can be seen from its interface, the navigation to its associated Port is possible, however Location cannot change or remove its Port (it is existentially dependent on it, therefore removing the bond would destroy the Location itself as well). Looking at Port (listing B.6), we see that it can change its Location (but not remove it as it is required in the model) and when it does so, the previous Location is invalidated to indicate it is no longer valid because of the existential dependency (see section 2.1.4).

11.1.7 Soft checking

Some of the constraints are enforced using soft checking (see section 10.2.10). A good example of this can be seen in Registrar code (listing B.7). In its *IsValid* method both a relation end with at most one member and an end with multiple members can be validated using the *deep* parameter.

The conditions are placed in such an order that the most expensive operations (recursive validation) are placed last. This ensures that if the cheap count checks fail, the expensive ones are not even performed.

11.2 Case study 2

The diagram of the second sample model is shown in fig. 11.2. There is a Person that can be in two Phases. The Person must have a Brain and a Living Person can be a member of various teams. Robots can also be members of teams and both Person and Robot must have a name, and so they belong to the NamedIndividual Category.

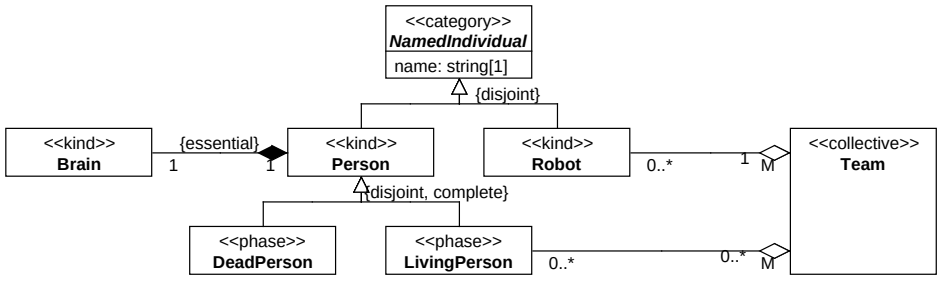


Figure 11.2: Case study 2

This model was created to cover the constructs not covered in Case study 1, so it has been simplified as much as possible (even removing attributes). The generated code is listed in Appendix C. The constructs covered here are:

- Phases
- Essential relations
- Special relations (memberOf)
- Non-sortals (Category)

11.2.1 Phases

As seen in the model, a Person must be either in DeadPerson or LivingPerson Phase. When covering Phases mapping (section 8.3.4) we discussed how this is achieved. The generated interface, *IPersonPhase*, is shown in listing C.4 and the generated Phases (both implementing the interface) in listing C.2 and listing C.6. The interface contains only one method – GetPerson. This enables instances of Phases to navigate to their owner and the absence of setter and unsetter makes sure that an instance of a Phase does not abandon its owner.

The Phase owner, Person, is listed in listing C.8. We can see that its constructor has a parameter of the IPersonPhase type. This (along with missing unset method for the Phase) ensures, that every Person is in some Phase.

11.2.2 Essential relations

There is an essential relation between Person and Brain. This means, that every Person must have a Brain since the moment it is created (see section 2.2.2). As we can see in listing C.8, an instance of IBrain must be passed to the Person's constructor. Moreover this instance is checked for nulls, throwing an exception if the check fails. Also, there is no setter or unsetter for the other member in both Person and Brain (listing C.1).

11.2.3 Special relations

In section 8.3.8 we stated that *memberOf*, *subCollectionOf* and *subQuantityOf* are handled similarly. In our model, Team (listing C.10) can have Person and Robot members. Note that these have different cardinality of their relation to Team. The generated interface, *ITeamMember*, is listed in listing C.5 and both Person and Robot implement it. Team then works with instances of that interface, not the Person or Robot directly.

11.2.4 Non-sortals

The *NamedIndividual* Category listed in listing C.7 is an example on how non-sortal types are handled (see section 2.1.3). It has only one property, Name. Robot as a member of the Category, implements its interface (see listing C.9), contains an instance of the Category and delegates the Name calls

to that instance. The need for the inner instance is the reason the *abstract* attribute is ignored in this implementation.

Chapter 12

Final assessment

In this chapter the overall performance (mainly in terms of preserving the fidelity of model semantics) is discussed. Also, areas for eventual further work are laid out.

12.1 Model fidelity

When discussing model fidelity I find that the most expressive way is to list all the aspects of the model that are preserved and those that are not. The aspects of the OntoUML model preserved in the generated C# code in at least a soft check form are:

- Entity names
- Entity attributes including cardinality constraints and type
- Inheritance structure even with overlapping inheritance
- Role and Phase semantics
- Part whole relations with cardinality and the `essential` and `inseparable` metaatributes
- *memberOf*, *subQuantityOf* and *subCollectionOf* relations without cardinality
- Derived relations (only the get part)
- Aspect types existential dependency property

These can be seen in the Case studies (see chapter 11). Parts that are not included:

- `immutablePart` and `immutableWhole` metaatributes

- *memberOf*, *subQuantityOf* and *subCollectionOf* relations cardinality
- *abstract* class attribute

The *immutable* attributes were not implemented in the C# code generator, because I was not able to invent a way to do it elegantly and efficiently enough. The cardinality of the enumerated relation types is lost in the OntoObjectModel creation for reasons discussed there (see section 8.3.8). Ignoring of the *abstract* by the C# implementation is explained in section 11.2.4.

12.2 Future work

There are a few parts of the approach that could be improved or expanded upon. Most importantly the missing aspects listed in the Model fidelity section (12.1) provide a topic to research further.

Other area that could be made better is naming of the generated entities and properties. The pluralization of the names is done naïvely and it could be made more grammatically correct (understandably, this was not this thesis goal). Another naming problem is in the classes generated from overlapping inheritance. As it stands their names are created simply by concatenating the names of the atomic classes. This can lead to very strange and in cases of wide inheritance trees even very long names. A solution could be to provide the user an option to specify the names manually (in the current state this can be fixed using a simple search and replace in the generated model but that is far from usable).

This is connected to another area that could be a research topic – integrating the application (or at least the described algorithms) into some modelling tool. This would bridge the gap between creating the model and generating the code. The last and most obvious area of future work is adding support for other input and output languages.

Conclusion

In Part I, Model-Driven Engineering and its implementation — Model-Driven Architecture — were presented (chapter 1). OntoUML as an useful language for conceptual modelling was introduced (chapter 2). Object Model as a theoretical basis for object-oriented programming was described (chapter 3) and the relevant features of the C# language were detailed as well as the way it implements the Object Model in chapter 4. Finally, existing languages that are used in OntoUML to source code transformation were reviewed (chapter 5).

The practical Part II introduced the transformations and intermediate models on the way from OntoUML model to C# source code. First the canonical OntoUML form was defined in chapter 7, then the OntoObjectModel structure and the algorithm to implement various OntoUML construct in it was described in chapter 8 and finally, the process of creating C# code from it was explained in chapters 9 and 10.

Part III is dedicated to evaluating the viability and suitability of the solution chosen. Simple case studies were created and their aspects discussed in chapter 11, and in chapter 12 the overall coverage of the OntoUML as well as opportunities for further work are examined.

I believe the presented solution proved to be usable for real life applications. It covers majority of OntoUML concepts while remaining reasonably simple. Also, this thesis can serve as a good starting point for following research in this field.

Bibliography

[1] Rosenwasser, D.; Najmabadi, C.; Hejlsberg, A.; et al. TypeScript - JavaScript that scales. 2016, [Accessed 2016-04-01]. Available from: https://www.typescriptlang.org/

[2] Dahl, R.; Noordhuis, B.; Schlueter, I. Z.; et al. Node.js. 2016, [Accessed 2016-04-01]. Available from: https://nodejs.org/en/

[3] Microsoft. Visual Studio Code [software]. [Accessed 2016-04-29]. Available from: https://code.visualstudio.com/

[4] Microsoft. Visual Studio 2015 [software]. [Accessed 2016-04-23]. Available from: https://www.visualstudio.com/en-us/products/vs-2015-product-editions.aspx

[5] Foundation, N. NuGet Gallery. 2016, [Accessed 2016-04-08]. Available from: https://www.nuget.org/

[6] Microsoft. .NET Framework 4.5.2 [software]. [Accessed 2016-04-23]. Available from: https://www.microsoft.com/en-us/download/details.aspx?id=42642

[7] Wilson, B.; et al. xUnit.net. [Accessed 2016-04-26]. Available from: https://github.com/xunit/xunit

[8] Sales, T. P.; Amorim, V.; Brasileiro, F. OntoUML lightweight editor (OLED) [software]. 2016, [Accessed 2016-01-21]. Available from: https://github.com/nemo-ufes/ontouml-lightweight-editor

[9] Auer, M.; Poelz, J.; Fuernweger, A.; et al. UMLet [software]. [Accessed 2016-04-23]. Available from: http://www.umlet.com/

[10] Shanu, U. Data flow diagrams. 2013, [Accessed 2016-04-23]. Available from: http://www.slideshare.net/ujjmishra1/data-flow-diagrams-2

[11] van der Zander, B.; Sundermeyer, J.; Braun, D.; et al. TeXstudio [software]. [Accessed 2016-04-23]. Available from: http://www.texstudio.org/

[12] Stecklein, J. M.; Dabney, J.; Dick, B.; et al. Error Cost Escalation Through the Project Life Cycle. [Accessed 2016-02-12]. Available from: http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20100036670.pdf

[13] Object Management Group ®. Object Management Group Model Driven Architecture (MDA) MDA Guide rev. 2.0. [Accessed 2016-01-23]. Available from: http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf

[14] Truyen, F. The Fast Guide to Model Driven Architecture. Jan 2006, [Accessed 2016-01-23]. Available from: http://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf

[15] Object Management Group ®. (MDA Foundation Model. [Accessed 2016-01-23]. Available from: http://www.omg.org/cgi-bin/doc?ormsc/10-09-06.pdf

[16] Guizzardi, G.; Wagner, G. Using the Unified Foundational Ontology (UFO) as a Foundation for General Conceptual Modeling Languages. In *Theory and Applications of Ontology: Computer Applications*, edited by R. Poli; M. Healy; A. Kameas, Springer Netherlands, 2010, ISBN 978-90-481-8846-8, 978-90-481-8847-5, pp. 175–196. Available from: http://link.springer.com/chapter/10.1007/978-90-481-8847-5_8

[17] American Psychological Association. Glossary of Psychological Terms. 2016, [Accessed 2016-03-31]. Available from: https://www.apa.org/research/action/glossary.aspx?tab=3

[18] Guizzardi, G. *Ontological Foundations For Structural Conceptual Models*. Enschede: Centre for Telematics and Information Technology, Telematica Instituut, 2005, ISBN 90-75176-81-3.

[19] Guizzardi, G.; Andrade Almeida, J.; Souza Guizzardi, R.; et al. Ontology & Conceptual Modeling Research Group (NEMO). 2014, [cit. 2014-04-23]. Available from: http://nemo.inf.ufes.br/en

[20] Wiegers, R. *Behavior Specification for Ontologically Grounded Conceptual Models*. Master's thesis, University of Twente, 2014.

[21] Pergl, R.; Sales, T. P.; Rybola, Z. Towards OntoUML for Software Engineering: From Domain Ontology to Implementation Model. In *Proceedings of MEDI 2013*, volume 3rd, Amantea, Italy: Springer, Sept. 2013, ISBN 978-3-642-41365-0, pp. 249–263, doi:10.1007/978-3-642-41366-7, 00001.

[22] Bruce, K. *Foundations of object-oriented languages types and semantics*. Cambridge, Mass: MIT Press, 2002, ISBN 978-0-262-02523-2.

[23] Microsoft. C# Language Specification 5.0. 2012, [Accessed 2016-03-30]. Available from: https://www.microsoft.com/en-us/download/details.aspx?id=7029

[24] Microsoft. Access Modifiers (C# Reference). 2016, [Accessed 2016-02-05]. Available from: https://msdn.microsoft.com/en-us/library/wxh6fsc7.aspx

[25] Michaelis, M. C# : The New and Improved C# 6.0. 2014, [Accessed 2016-04-22]. Available from: https://msdn.microsoft.com/en-us/magazine/dn802602.aspx

[26] Schreiber, G.; Raimond, Y. RDF 1.1 Primer. [Accessed 2016-01-21]. Available from: https://www.w3.org/TR/rdf11-primer/

[27] Hitzler, P.; Krötzsch, M.; Parsia, B.; et al. OWL 2 Web Ontology Language Primer (Second Edition). [Accessed 2016-01-21]. Available from: https://www.w3.org/TR/2012/REC-owl2-primer-20121211/

[28] Motik, B.; Grau, B. C.; Horrocks, I.; et al. OWL 2 Web Ontology Language Profiles (Second Edition)). [Accessed 2016-01-21]. Available from: https://www.w3.org/TR/owl2-profiles/

[29] ROWLEX Admin. ROWLEX . Net RDF website access? [Accessed 2016-02-05]. Available from: http://stackoverflow.com/a/7038336/2546338

[30] Szekely, B.; Betz, J. Jastor – Typesafe, Ontology Driven RDF Access from Java. [Accessed 2016-02-05]. Available from: http://jastor.sourceforge.net/

[31] Szekely, B.; Betz, J. Jastor – Browse Files at SourceForge.net. [Accessed 2016-02-05]. Available from: http://sourceforge.net/projects/jastor/files/

[32] Musen, M.; Tu, S.; Tudorache, T.; et al. protégé. Available from: http://protege.stanford.edu/

[33] Musen, M.; Tu, S.; Tudorache, T.; et al. Protege Desktop Older Versions. Available from: http://protegewiki.stanford.edu/wiki/Protege_Desktop_Old_Versions

[34] Braga, B.; Brasileiro, F.; Guerson, J.; et al. Menthor. 2015, [Accessed 2016-04-23]. Available from: http://www.menthor.net

[35] Sales, T. P.; Amorim, V.; Brasileiro, F. OntoUML lightweight editor (OLED) Releases. 2016, [Accessed 2016-03-30]. Available from: https://github.com/nemo-ufes/ontouml-lightweight-editor/releases

[36] Foundation, N. Download — Node.js. 2016, [Accessed 2016-04-01]. Available from: https://nodejs.org/en/download/stable/

[37] DeBill, E. Modulecounts. 2016, [Accessed 2016-04-01]. Available from: http://www.modulecounts.com/

[38] Brookes, L. command-line-args. 2016, [Accessed 2016-04-01]. Available from: https://www.npmjs.com/package/command-line-args

[39] Katz, Y.; Decker, K.; Johnson, A.; et al. handlebars. 2016, [Accessed 2016-04-01]. Available from: https://www.npmjs.com/package/handlebars

[40] Estácio, L. json-format. 2016, [Accessed 2016-04-01]. Available from: https://www.npmjs.com/package/json-format

[41] Dalton, J.; Ashkenas, J.; et al. lodash. 2016, [Accessed 2016-04-01]. Available from: https://www.npmjs.com/package/lodash

[42] Halliday, J. mkdirp. 2016, [Accessed 2016-04-01]. Available from: https://www.npmjs.com/package/mkdirp

[43] Kowal, K.; Denicola, D. q. 2016, [Accessed 2016-04-01]. Available from: https://www.npmjs.com/package/q

[44] Luff, G. tv4. 2016, [Accessed 2016-04-01]. Available from: https://www.npmjs.com/package/tv4

[45] Kubica, M. xml2js. 2016, [Accessed 2016-04-01]. Available from: https://www.npmjs.com/package/xml2js

[46] Luer, J.; Todorov, V.; et al. chai. 2016, [Accessed 2016-04-21]. Available from: https://www.npmjs.com/package/chai

[47] Denicola, D. chai-as-promised. 2016, [Accessed 2016-04-21]. Available from: https://www.npmjs.com/package/chai-as-promised

[48] Anantheswaran, K. istanbul. 2016, [Accessed 2016-04-21]. Available from: https://www.npmjs.com/package/istanbul

[49] Holowaychuk, T.; Jeffery, T.; Hiller, C.; et al. mocha. 2016, [Accessed 2016-04-21]. Available from: https://www.npmjs.com/package/mocha

[50] Schaub, T. mock-fs. 2016, [Accessed 2016-04-21]. Available from: https://www.npmjs.com/package/mock-fs

[51] Kelly, K. remap-istanbul. 2016, [Accessed 2016-04-21]. Available from: https://www.npmjs.com/package/remap-istanbul

[52] Rosenwasser, D.; Najmabadi, C.; Hejlsberg, A.; et al. typescript. 2016, [Accessed 2016-04-21]. Available from: https://www.npmjs.com/package/typescript

[53] Embrey, B.; et al. typings/typings: The TypeScript Definition Manager. Jun 2016, [Accessed 2016-04-01]. Available from: https://github.com/typings/typings

[54] ECMA. ECMAScript® 2015 Language Specification. Jun 2015, [Accessed 2016-04-01]. Available from: http://www.ecma-international.org/ecma-262/6.0/

[55] IEFT. JSON Schema: core definitions and terminology. 2016, [Accessed 2016-04-08]. Available from: https://tools.ietf.org/html/draft-zyp-json-schema-04

[56] Network, M. D. Promise - JavaScript — MDN. 2016, [Accessed 2016-04-03]. Available from: https://developer.mozilla.org/cs/docs/Web/JavaScript/Reference/Global_Objects/Promise

[57] Nishizawa, K. Memory Leak? native Promise and complex arrow functions with Generators. 2015, [Accessed 2016-04-24]. Available from: https://github.com/nodejs/node/issues/4210

[58] Homola, D. ontouml-code-generator: Release for thesis. May 2016, doi: 10.5281/zenodo.50713, [Accessed 2016-05-01]. Available from: http://dx.doi.org/10.5281/zenodo.50713

[59] Homola, D. ontouml-csharp-utils: Release for thesis. May 2016, doi: 10.5281/zenodo.50714, [Accessed 2016-05-01]. Available from: http://dx.doi.org/10.5281/zenodo.50714

[60] Homola, D. ontouml-utilities MyGet Feed. May 2016, [Accessed 2016-05-01]. Available from: https://www.myget.org/F/ontouml-utilities

[61] Dominik Gessenharter. Implementing UML associations in Java: a slim code pattern for a complex modeling concept. In *Proceedings of the Workshop on Relationships and Associations in Object-Oriented Languages*, RAOOL '09, New York, NY, USA: ACM, 2009, ISBN 978-1-60558-549-9, pp. 17–24, doi:10.1145/1562100.1562104, 00008. Available from: http://doi.acm.org/10.1145/1562100.1562104

[62] Microsoft. Lazy(T) Class (System). 2016, [Accessed 2016-03-21]. Available from: https://msdn.microsoft.com/en-us/library/dd642331(v=vs.110).aspx

[63] Microsoft. System.Collections.Generic Namespace. 2016, [Accessed 2016-03-21]. Available from: https://msdn.microsoft.com/en-us/library/system.collections.generic(v=vs.110).aspx

[64] Microsoft. List(T) Class (System.Collections.Generic). 2016, [Accessed 2016-03-21]. Available from: https://msdn.microsoft.com/en-us/library/6sh2ey19(v=vs.110).aspx

[65] Sales, T. P. Role. 2016, [Accessed 2016-04-04]. Available from: https://github.com/MenthorTools/ontouml/wiki/Role

[66] Microsoft. Value Types Table (C# Reference). 2016, [Accessed 2016-04-13]. Available from: https://msdn.microsoft.com/en-us/library/bfft1t3c.aspx

[67] Two, M. Difference Between Select and SelectMany. 2009, [Accessed 2016-04-22]. Available from: http://stackoverflow.com/a/959057/2546338

[68] Microsoft. Enumerable Methods. 2016, [Accessed 2016-03-26]. Available from: https://msdn.microsoft.com/en-us/library/system.linq.enumerable_methods(v=vs.100).aspx

Appendix A

Acronyms

CIM Computation Independent Model

DEMO Design & Engineering Methodology for Organizations

JSON JavaScript Object Notation

MDA Model-Driven Architecture

MDE Model-Driven Engineering

NEMO Núcleo de Estudos em Modelagem Conceitual e Ontologias (Ontology and Conceptual Modeling Research Group)

OntoObjectModel Ontological Object Model

OLED OntoUML Lightweight Editor

OMG Object Management Group ®

OWL Web Ontology Language

PIM Platform Independent Model

PSM Platform Specific Model

RDF Resource Description Framework

UML Unified Modelling Language

W3C World Wide Web Consortium

XMI XML Metadata Interchange

XML Extensible Markup Language

XSD XML Schema Definition

Appendix B

Case study 1 generated code

```
public interface IBoat : ICanValidate, IVehicle
{
  void SetRegistration(IRegistration registration);
  void UnsetRegistration();
  IRegistration GetRegistration();
  IRegistrar GetRegistrar();
}
public class Boat : Vehicle, IBoat
{
  public Boat()
  {  }
  private static OneToOneAssociation<IRegistration, IBoat>
    boatRegistrationAssociation
    = OneToOneAssociation<IRegistration, IBoat>.Instance();
  public void SetRegistration(IRegistration registration)
  {
    UnsetRegistration();
    boatRegistrationAssociation
      .CreateLink(registration, this);
  }
  public void UnsetRegistration()
  {
    boatRegistrationAssociation
      .DestroyLink(boatRegistrationAssociation
        .GetLeft(this), this);
  }
  public IRegistration GetRegistration()
    => boatRegistrationAssociation.GetLeft(this);
  public IRegistrar GetRegistrar()
```

```
    => GetRegistration()?.GetRegistrar();
  private bool isInvalidated = false;
  public override void Invalidate()
  {
    isInvalidated = true;
  }
  public override bool IsValid(bool deep)
  {
    if (isInvalidated) return false;
    var registrationToValidate = GetRegistration();
    if (boatRegistrationAssociation == null
      || (deep && !registrationToValidate.IsValid(deep)))
      return false;
    return true;
  }
}
```

Listing B.1: Boat.cs

```
public interface IBoatPlane : ICanValidate, IBoat, IPlane
{
}
public class BoatPlane : Vehicle, IBoatPlane
{
  public BoatPlane()
  {
    boat = new Boat();
    plane = new Plane();
  }
  private readonly IBoat boat;
  private readonly IPlane plane;
  private string name;
  public override string Name
  {
    get { return name; }
    set
    {
      name = value;
      boat.Name = value;
      plane.Name = value;
    }
  }
  public IBoundedCollection<string> CallSigns
  {
    get { return plane.CallSigns; }
```

```
    set { plane.CallSigns = value; }
  }
  public void SetRegistration(IRegistration registration)
    => boat.SetRegistration(registration);
  public void UnsetRegistration()
    => boat.UnsetRegistration();
  public IRegistration GetRegistration()
    => boat.GetRegistration();
  public IRegistrar GetRegistrar()
    => boat.GetRegistrar();
  private bool isInvalidated = false;
  public override void Invalidate()
  {
    isInvalidated = true;
  }
  public override bool IsValid(bool deep)
  {
    if (isInvalidated) return false;
    if (boat == null || !boat.IsValid(deep))
      return false;
    if (plane == null || !plane.IsValid(deep))
      return false;
    return true;
  }
```

Listing B.2: BoatPlane.cs

```
public interface ICanValidate
{
  bool IsValid(bool deep);
  void Invalidate();
}
```

Listing B.3: ICanValidate.cs

```
public interface ILocation : ICanValidate
{
  decimal? Longitude { get; set; }
  decimal? Latitude { get; set; }
  IPort GetPort();
}
public class Location : ILocation
{
  public Location()
  {  }
```

```
  private static OneToOneAssociation<ILocation, IPort>
    portLocationAssociation
    = OneToOneAssociation<ILocation, IPort>.Instance();
  public IPort GetPort()
    => portLocationAssociation.GetRight(this);
  private decimal? longitude;
  public virtual decimal? Longitude
  {
    get { return longitude; }
    set { longitude = value; }
  }
  private decimal? latitude;
  public virtual decimal? Latitude
  {
    get { return latitude; }
    set { latitude = value; }
  }
  private bool isInvalidated = false;
  public virtual void Invalidate()
  {
    isInvalidated = true;
  }
  public virtual bool IsValid(bool deep)
  {
    if (isInvalidated) return false;
    var portToValidate = GetPort();
    if (portLocationAssociation == null
      || portToValidate == null
      || (deep && !portToValidate.IsValid(deep)))
      return false;
    if (Longitude == null) return false;
    if (Latitude == null) return false;
    return true;
  }
}
```

Listing B.4: Location.cs

```
public interface IPlane : ICanValidate, IVehicle
{
  IBoundedCollection<string> CallSigns { get; set; }
}
public class Plane : Vehicle, IPlane
{
  public Plane()
  {  }
  private IBoundedCollection<string> callSigns;
  public virtual IBoundedCollection<string> CallSigns
  {
    get { return callSigns; }
    set
    {
      if (value != null)
      {
        const int MAX_ITEMS = 5;
        if (value.MaxItems > MAX_ITEMS)
          throw new InvalidOperationException(
        "The source collection for CallSigns is too small."
          );
        value.MaxItems = MAX_ITEMS;
      }
      callSigns = value;
    }
  }
  private bool isInvalidated = false;
  public override void Invalidate()
  {
    isInvalidated = true;
  }
  public override bool IsValid(bool deep)
  {
    if (isInvalidated) return false;
    const int CALL_SIGNS_MIN_ITEMS = 1;
    if (CallSigns == null
      || CallSigns.Count() < CALL_SIGNS_MIN_ITEMS)
      return false;
    return true;
  }
}
```

Listing B.5: Plane.cs

```
public interface IPort : ICanValidate
{
  void AddRegistrarRole(IRegistrar registrarRole);
  void RemoveRegistrarRole(IRegistrar registrarRole);
  IEnumerable<IRegistrar> GetRegistrarRoles();
  void SetLocation(ILocation location);
  ILocation GetLocation();
}
public class Port : IPort
{
  public Port()
  {  }
  private static OneToManyAssociation<IPort, IRegistrar>
    registrarRolesAssociation
    = OneToManyAssociation<IPort, IRegistrar>
      .Instance(true);
  public void AddRegistrarRole(IRegistrar registrarRole)
    => registrarRolesAssociation
      .CreateLink(this, registrarRole);
  public void RemoveRegistrarRole(IRegistrar registrarRole)
  {
    registrarRole?.Invalidate();
    registrarRolesAssociation
      .DestroyLink(this, registrarRole);
  }
  public IEnumerable<IRegistrar> GetRegistrarRoles()
    => registrarRolesAssociation.GetMany(this);
  private static OneToOneAssociation<ILocation, IPort>
    portLocationAssociation
    = OneToOneAssociation<ILocation, IPort>.Instance();
  public void SetLocation(ILocation location)
  {
    UnsetLocation();
    portLocationAssociation.CreateLink(location, this);
  }
  private void UnsetLocation()
  {
    GetLocation()?.Invalidate();
    portLocationAssociation
      .DestroyLink(portLocationAssociation
        .GetLeft(this), this);
  }
  public ILocation GetLocation()
    => portLocationAssociation.GetLeft(this);
```

```
  private bool isInvalidated = false;
  public virtual void Invalidate()
  {
    isInvalidated = true;
    foreach (var item in GetRegistrarRoles())
    {
      RemoveRegistrarRole(item);
    }
    UnsetLocation();
  }
  public virtual bool IsValid(bool deep)
  {
    if (isInvalidated) return false;
    var registrarRoleToValidate = GetRegistrarRoles();
    if (registrarRolesAssociation == null
      || (deep && registrarRoleToValidate
        .Any(x => !x.IsValid(deep))))
      return false;
    var locationToValidate = GetLocation();
    if (portLocationAssociation == null
      || locationToValidate == null
      || (deep && !locationToValidate.IsValid(deep)))
      return false;
    return true;
  }
}
```

Listing B.6: Port.cs

```
public interface IRegistrar : ICanValidate
{
  IPort GetPort();
  void AddRegistration(IRegistration registration);
  void RemoveRegistration(IRegistration registration);
  IEnumerable<IRegistration> GetRegistrations();
  IEnumerable<IBoat> GetBoats();
}
public class Registrar : IRegistrar
{
  public Registrar()
  {  }
  private static OneToManyAssociation<IPort, IRegistrar>
    registrarRolesAssociation
    = OneToManyAssociation<IPort, IRegistrar>
      .Instance(true);
  public IPort GetPort()
    => registrarRolesAssociation.GetOne(this);
  private static OneToManyAssociation
    <IRegistrar, IRegistration>
    registrarRegistrationAssociation
    = OneToManyAssociation<IRegistrar, IRegistration>
      .Instance();
  public void AddRegistration(IRegistration registration)
    => registrarRegistrationAssociation
       .CreateLink(this, registration);
  public void RemoveRegistration(IRegistration registration)
  {
    registrarRegistrationAssociation
      .DestroyLink(this, registration);
  }
  public IEnumerable<IRegistration> GetRegistrations()
    => registrarRegistrationAssociation.GetMany(this);
  public IEnumerable<IBoat> GetBoats()
    => GetRegistrations()
    .Select(r => r.GetBoat()).Distinct();
  private bool isInvalidated = false;
  public virtual void Invalidate()
  {
    isInvalidated = true;
  }
  public virtual bool IsValid(bool deep)
  {
    if (isInvalidated) return false;
```

```
    var portToValidate = GetPort();
    if (registrarRolesAssociation == null
      || portToValidate == null
      || (deep && !portToValidate.IsValid(deep)))
      return false;
    const int REGISTRATION_MIN_ITEMS = 1;
    var registrationToValidate = GetRegistrations();
    var registrationCount = registrationToValidate.Count();
    if (registrarRegistrationAssociation == null
      || registrationCount < REGISTRATION_MIN_ITEMS
      || (deep && registrationToValidate
        .Any(x => !x.IsValid(deep))))
      return false;
    return true;
  }
}
```

Listing B.7: Registrar.cs

```
public interface IRegistration : ICanValidate
{
  void SetBoat(IBoat boat);
  IBoat GetBoat();
  void SetRegistrar(IRegistrar registrar);
  IRegistrar GetRegistrar();
}
public class Registration : IRegistration
{
  public Registration()
  { }
  private static OneToOneAssociation<IRegistration, IBoat>
    boatRegistrationAssociation
    = OneToOneAssociation<IRegistration, IBoat>.Instance();
  public void SetBoat(IBoat boat)
  {
    UnsetBoat();
    boatRegistrationAssociation.CreateLink(this, boat);
  }
  private void UnsetBoat()
  {
    boatRegistrationAssociation
      .DestroyLink(this, boatRegistrationAssociation
        .GetRight(this));
  }
  public IBoat GetBoat()
```

```
    => boatRegistrationAssociation.GetRight(this);
  private static OneToManyAssociation
    <IRegistrar, IRegistration>
    registrarRegistrationAssociation
    = OneToManyAssociation<IRegistrar, IRegistration>
      .Instance();
  public void SetRegistrar(IRegistrar registrar)
  {
    UnsetRegistrar();
    registrarRegistrationAssociation
      .CreateLink(registrar, this);
  }
  private void UnsetRegistrar()
  {
    registrarRegistrationAssociation
      .DestroyLink(registrarRegistrationAssociation
        .GetOne(this), this);
  }
  public IRegistrar GetRegistrar()
    => registrarRegistrationAssociation.GetOne(this);
  private bool isInvalidated = false;
  public virtual void Invalidate()
  {
    isInvalidated = true;
  }
  public virtual bool IsValid(bool deep)
  {
    if (isInvalidated) return false;
    var boatToValidate = GetBoat();
    if (boatRegistrationAssociation == null
      || boatToValidate == null
      || (deep && !boatToValidate.IsValid(deep)))
      return false;
    var registrarToValidate = GetRegistrar();
    if (registrarRegistrationAssociation == null
      || registrarToValidate == null
      || (deep && !registrarToValidate.IsValid(deep)))
      return false;
    return true;
  }
}
```

Listing B.8: Registration.cs

```
public interface IVehicle : ICanValidate
{
  string Name { get; set; }
}
public class Vehicle : IVehicle
{
  public Vehicle()
  {  }
  private string name;
  public virtual string Name
  {
    get { return name; }
    set { name = value; }
  }
  private bool isInvalidated = false;
  public virtual void Invalidate()
  {
    isInvalidated = true;
  }
  public virtual bool IsValid(bool deep)
  {
    if (isInvalidated) return false;
    if (Name == null) return false;
    return true;
  }
}
```

Listing B.9: Vehicle.cs

Appendix C

Case study 2 generated code

```
public interface IBrain : ICanValidate
{
  IPerson GetPerson();
}
public class Brain : IBrain
{
  public Brain() {  }
  private static OneToOneAssociation<IPerson, IBrain>
    personBrainAssociation
    = OneToOneAssociation<IPerson, IBrain>.Instance();
  public IPerson GetPerson()
    => personBrainAssociation.GetLeft(this);
  private bool isInvalidated = false;
  public virtual void Invalidate()
  {
    isInvalidated = true;
  }
  public virtual bool IsValid(bool deep)
  {
    if (isInvalidated) return false;
    var personToValidate = GetPerson();
    if (personBrainAssociation == null
      || personToValidate == null
      || (deep && !personToValidate.IsValid(deep)))
      return false;
    return true;
  }
}
```

Listing C.1: Brain.cs

```
public interface IDeadPerson : ICanValidate, IPersonPhase
{
}
public class DeadPerson : IDeadPerson
{
  public DeadPerson()
  {  }
  private static OneToOneAssociation<IPerson, IPersonPhase>
    personPhaseAssociation
    = OneToOneAssociation<IPerson, IPersonPhase>
      .Instance();
  public IPerson GetPerson()
    => personPhaseAssociation.GetLeft(this);
  private bool isInvalidated = false;
  public virtual void Invalidate()
  {
    isInvalidated = true;
  }
  public virtual bool IsValid(bool deep)
  {
    if (isInvalidated) return false;
    return true;
  }
}
```

Listing C.2: DeadPerson.cs

```
public interface ICanValidate
{
  bool IsValid(bool deep);
  void Invalidate();
}
```

Listing C.3: ICanValidate.cs

```
public interface IPersonPhase : ICanValidate
{
  IPerson GetPerson();
}
```

Listing C.4: IPersonPhase.cs

```
public interface ITeamMember : ICanValidate
{
  void AddTeam(ITeam team);
  void RemoveTeam(ITeam team);
```

```
  IEnumerable<ITeam> GetTeams();
}
```

Listing C.5: ITeamMember.cs

```
public interface ILivingPerson
  : ICanValidate, IPersonPhase, ITeamMember
{
}
public class LivingPerson : ILivingPerson
{
  public LivingPerson()
  {  }
  private static OneToOneAssociation<IPerson, IPersonPhase>
    personPhaseAssociation
    = OneToOneAssociation<IPerson, IPersonPhase>
      .Instance();
  public IPerson GetPerson()
    => personPhaseAssociation.GetLeft(this);
  private static ManyToManyAssociation<ITeam, ITeamMember>
    teamMembersAssociation
    = ManyToManyAssociation<ITeam, ITeamMember>
      .Instance();
  public void AddTeam(ITeam team)
    => teamMembersAssociation.CreateLink(team, this);
  public void RemoveTeam(ITeam team)
  {
    teamMembersAssociation.DestroyLink(team, this);
  }
  public IEnumerable<ITeam> GetTeams()
    => teamMembersAssociation.GetMs(this);
  private bool isInvalidated = false;
  public virtual void Invalidate()
  {
    isInvalidated = true;
  }
  public virtual bool IsValid(bool deep)
  {
    if (isInvalidated) return false;
    return true;
  }
}
```

Listing C.6: LivingPerson.cs

```
public interface INamedIndividual : ICanValidate
{
  string Name { get; set; }
}
public class NamedIndividual : INamedIndividual
{
  public NamedIndividual()
  {  }
  private string name;
  public virtual string Name
  {
    get { return name; }
    set { name = value; }
  }
  private bool isInvalidated = false;
  public virtual void Invalidate()
  {
    isInvalidated = true;
  }
  public virtual bool IsValid(bool deep)
  {
    if (isInvalidated) return false;
    if (Name == null) return false;
    return true;
  }
}
```

Listing C.7: NamedIndividual.cs

```
public interface IPerson : ICanValidate, INamedIndividual
{
  void SetPersonPhase(IPersonPhase personPhase);
  IPersonPhase GetPersonPhase();
  IBrain GetBrain();
}
public class Person : IPerson
{
  public Person(IPersonPhase personPhase, IBrain brain)
  {
    if (personPhase == null)
      throw new ArgumentNullException(nameof(personPhase));
    if (brain == null)
      throw new ArgumentNullException(nameof(brain));
    namedIndividual = new NamedIndividual();
    personPhaseAssociation.CreateLink(this, personPhase);
```

```
    personBrainAssociation.CreateLink(this, brain);
  }
  private static OneToOneAssociation<IPerson, IPersonPhase>
    personPhaseAssociation
    = OneToOneAssociation<IPerson, IPersonPhase>
      .Instance();
  public void SetPersonPhase(IPersonPhase personPhase)
  {
    UnsetPersonPhase();
    personPhaseAssociation.CreateLink(this, personPhase);
  }
  private void UnsetPersonPhase()
  {
    GetPersonPhase()?.Invalidate();
    personPhaseAssociation
      .DestroyLink(this, personPhaseAssociation
        .GetRight(this));
  }
  public IPersonPhase GetPersonPhase()
    => personPhaseAssociation.GetRight(this);
  private static OneToOneAssociation<IPerson, IBrain>
    personBrainAssociation
    = OneToOneAssociation<IPerson, IBrain>.Instance();
  public IBrain GetBrain()
    => personBrainAssociation.GetRight(this);
  private readonly INamedIndividual namedIndividual;
  public string Name
  {
    get { return namedIndividual.Name; }
    set { namedIndividual.Name = value; }
  }
  private bool isInvalidated = false;
  public virtual void Invalidate()
  {
    isInvalidated = true;
    UnsetPersonPhase();
  }
  public virtual bool IsValid(bool deep)
  {
    if (isInvalidated) return false;
    if (namedIndividual == null
      || !namedIndividual.IsValid(deep))
      return false;
    var personPhaseToValidate = GetPersonPhase();
```

```
    if (personPhaseAssociation == null
      || personPhaseToValidate == null
      || (deep && !personPhaseToValidate.IsValid(deep)))
      return false;
    var brainToValidate = GetBrain();
    if (personBrainAssociation == null
      || brainToValidate == null
      || (deep && !brainToValidate.IsValid(deep)))
      return false;
    return true;
  }
}
```

Listing C.8: Person.cs

```
public interface IRobot
  : ICanValidate, INamedIndividual, ITeamMember
{
}
public class Robot : IRobot
{
  public Robot()
  {
    namedIndividual = new NamedIndividual();
  }
  private readonly INamedIndividual namedIndividual;
  public string Name
  {
    get { return namedIndividual.Name; }
    set { namedIndividual.Name = value; }
  }
  private static ManyToManyAssociation<ITeam, ITeamMember>
    teamMembersAssociation
    = ManyToManyAssociation<ITeam, ITeamMember>
      .Instance();
  public void AddTeam(ITeam team)
    => teamMembersAssociation.CreateLink(team, this);
  public void RemoveTeam(ITeam team)
  {
    teamMembersAssociation.DestroyLink(team, this);
  }
  public IEnumerable<ITeam> GetTeams()
    => teamMembersAssociation.GetMs(this);
  private bool isInvalidated = false;
  public virtual void Invalidate()
```

```
  {
    isInvalidated = true;
  }
  public virtual bool IsValid(bool deep)
  {
    if (isInvalidated) return false;
    if (namedIndividual == null
      || !namedIndividual.IsValid(deep))
      return false;
    return true;
  }
}
```

Listing C.9: Robot.cs

```
public interface ITeam : ICanValidate
{
  void AddTeamMember(ITeamMember teamMember);
  void RemoveTeamMember(ITeamMember teamMember);
  IEnumerable<ITeamMember> GetTeamMembers();
}
public class Team : ITeam
{
  public Team()
  {  }
  private static ManyToManyAssociation<ITeam, ITeamMember>
    teamMembersAssociation
    = ManyToManyAssociation<ITeam, ITeamMember>
      .Instance();
  public void AddTeamMember(ITeamMember teamMember)
    => teamMembersAssociation.CreateLink(this, teamMember);
  public void RemoveTeamMember(ITeamMember teamMember)
  {
    teamMembersAssociation.DestroyLink(this, teamMember);
  }
  public IEnumerable<ITeamMember> GetTeamMembers()
    => teamMembersAssociation.GetNs(this);
  private bool isInvalidated = false;
  public virtual void Invalidate()
  {
    isInvalidated = true;
  }
  public virtual bool IsValid(bool deep)
  {
    if (isInvalidated) return false;
```

```
      var teamMemberToValidate = GetTeamMembers();
      if (teamMembersAssociation == null
        || (deep && teamMemberToValidate
            .Any(x => !x.IsValid(deep))))
        return false;
      return true;
    }
}
```

Listing C.10: Team.cs

Appendix D

Contents of enclosed DVD

- `readme.txt` ..... brief DVD contents description
- `impl` ..... implementation files
  - `code-generator` ..... code generator application and its source codes
    - `examples` ..... source files for the Case studies
  - `csharp-utils` ..... C# utility library and source codes
- `src` ..... thesis source files
  - `DP_Homola_Dan_2016.tex` ..... thesis LaTeX source
- `text` ..... thesis text
  - `DP_Homola_Dan_2016.pdf` ..... thesis text in PDF