

Model-based interface code generation : an extensible framework for automatic generation of wrapper code

Citation for published version (APA):

Dalaikhuu, S. (2017). *Model-based interface code generation : an extensible framework for automatic generation of wrapper code*. Technische Universiteit Eindhoven.

Document status and date:

Published: 28/09/2017

Document Version:

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy

If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

/ Department of
Mathematics and
Computer Science
/ PDEng Software
Technology

Model-Based Interface Code Generation:

An extensible framework for automatic
generation of wrapper code

Sodkhuu Dalaikhuu

Model-Based Interface Code Generation

An extensible framework for automatic generation of wrapper code

Sodkhuu Dalaikhuu

Eindhoven University of Technology
Stan Ackermans Institute / Software Technology

Partners

ThermoFisher
S C I E N T I F I C

TU/e

Technische Universiteit
Eindhoven
University of Technology

Thermo Fisher Scientific Eindhoven University of
Technology

Steering Group

Martijn Kabel & Arjen Klomp (Project Managers)
E.P.H. de Groot (Project Mentors)
Andrei Radulescu & E.P. de Vink (Project Supervisors)

Date

13 September 2017

Document Status

Public

PDEng report no.

PDEng thesis ; 2017/050

The design described in this report has been carried out in accordance with TU/e Code of Scientific Conduct.

Contact Address	Eindhoven University of Technology Department of Mathematics and Computer Science MF 5.080A, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands +31402474334
Published by	Eindhoven University of Technology Stan Ackermans Institute
Printed by	Eindhoven University of Technology <i>UniversiteitsDrukkerij</i>
PDEng Report No.	PDEng thesis ; 2017/050
Abstract	Implementation and maintenance of interface wrapper code are relatively mechanical tasks. These tasks are repetitive, laborious, and error prone including numerous copy-paste actions and manual modifications from previous implementation samples. This report describes design, implementation, and deployment of an extensible framework for automatic generation of wrapper code. The framework, which is based on the Model-Driven Architecture, consists of parser, model-to-model transformer, and model-to-code transformer. The code generation framework described in this report is more suitable for using in a C# .NET environment than other existing technologies like Eclipse xtext/xtend. It is extensible for multiple domain specific languages and code artifacts, as well as simple to use with Visual Studio tool.
Keywords	Model-Based Development, Model-Driven Architecture, wrapper code, automatic code generation, Domain Specific Language, model transformation, Component Object Model, Interface Definition Language, software technology, PDEng
Preferred Reference	<u>MODEL-BASED INTERFACE CODE GENERATION: AN EXTENSIBLE FRAMEWORK FOR AUTOMATIC GENERATION OF WRAPPER CODE</u> , SAI Technical Report, September 2017. (PDEng thesis ; 2017/050)
Partnership	This project was supported by Eindhoven University of Technology and Thermo Fisher Scientific.
Disclaimer Endorsement	Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the Eindhoven University of Technology or Thermo Fisher Scientific. The views and opinions of authors expressed herein do not necessarily state or reflect those of the Eindhoven University of Technology or Thermo Fisher Scientific, and shall not be used for advertising or product endorsement purposes.

Disclaimer Liability	While every effort will be made to ensure that information contained within this report is accurate and up to date, Eindhoven University of Technology makes no warranty, representation, or undertaking whether expressed or implied, nor does it assume any legal liability, whether direct or indirect, or responsibility for the accuracy, completeness, or usefulness of any information.
Trademarks	Product and company names mentioned herein may be trademarks and/or services marks of their respective owners. We use these name without any particular endorsement or with the intent to infringe the copyright of the respective owners.
Copyright	Copyright © 2017. Eindhoven University of Technology. All rights reserved. No part of the material protected by this copyright notice may be reproduced, modified, or redistributed in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the Eindhoven University of Technology and Thermo Fisher Scientific.

Abstract

Implementation and maintenance of interface wrapper code are relatively mechanical tasks. These tasks are repetitive, laborious, and error prone including numerous copy-paste actions and manual modifications from previous implementation samples. This report describes design, implementation, and deployment of an extensible framework for automatic generation of wrapper code. The framework, which is based on the Model-Driven Architecture, consists of parser, model-to-model transformer, and model-to-code transformer. The code generation framework described in this report is more suitable for using in a C# .NET environment than other existing technologies like Eclipse xtext/xtend. It is extensible for multiple domain specific languages and code artifacts, as well as simple to use with Visual Studio tool.

Foreword

Over the last couple of years, a trend was forming in the software department of Thermo Fisher Scientific (formerly FEI). Several teams have investigated the use of code generation to reduce repetitive error-prone manual work. A number of approaches and different tools were applied with various degrees of success. The time was ripe to consolidate the past efforts and invest in a department global code generation framework. Sodkhuu has spent his time at Thermo Fisher to design and implement a code generation framework based on the requirements that we gathered from previous projects. From the start, he kept focus on delivering a framework that fits in our development process and can be used with minimal effort. Thanks to this focus, Sodkhuu managed to deliver a framework that is now part of our software code base and build infrastructure and is being used to generate files that were written manually before. Whereas other efforts mostly remained at a prototype level, Sodkhuu's framework is now operational in our build infrastructure. Currently the framework is only applied to one pilot project and only a few files are generated. However, the framework provides the flexibility to quickly extend the capabilities onto other projects. Now that the framework has proven itself to work, I have no doubt that several teams will eagerly start using it within their own projects. Eventually the amount of (manually written) code will be significantly reduced, which will lead to more stable releases, less maintenance effort and faster feature development. I would like to thank Sodkhuu for his valuable contribution and wish him all the best with his future endeavors.

Dr. ir. E.P.H. de Groot
Project Mentor
9 September 2017

Preface

This document summarizes the “Model-Based Interface Code Generation: An extensible framework for automatic generation of wrapper code” project. The project addresses the challenge of generating wrapper code using model-based technology and applying into a high-complex software system.

The project was executed by Sodkhuu Dalaikhuu from the Stan Ackerman’s Institute, PDEng Software Technology program of the Eindhoven University of Technology. This project is the nine-month final assignment for the aforementioned two-year Professional Doctorate in Engineering (PDEng) program, known by its Dutch name as Ontwerpers Opleiding Technische Informatica (OOTI). This project was implemented within Thermo Fisher Scientific company in Eindhoven.

This document is primarily intended for readers with a technical background in disciplines, such as automation, model-based technologies, wrapper code, domain specific languages, derivative text based languages, and general software engineering. However, no specialized knowledge in these disciplines is needed.

Readers with a non-technical background or who are interested in knowing the basis and results of this project should read chapters 1 - 3 and 8 - 10. These chapters introduce and explain the essential points in which the framework is aimed and the results that we have obtained with this project.

Readers who are interested in the detailed technical solution of the project should read chapters 4 - 7. These chapters cover design, implementation, deployment, as well as verification and validation phases of the project.

Sodkhuu Dalaikhuu
9 September 2017

Acknowledgments

This thesis would not have been possible without the inspiration and support of a number of wonderful individuals. My thanks and appreciation to all of them for being part of this journey and making this thesis possible.

I am deeply grateful with the people from Thermo Fisher Scientific company. Special thanks to Andrei Radulescu and Erwin de Groot who have been my supervisor and mentor during this project at the company. Their continuous support, motivation, collaboration, and feedback have been essential for the success of this project. I have enjoyed working with you during the nine months and thank you very much for the great experience that has helped me to further develop myself professionally. I would also like to thank to all Thermo Fisher Scientific engineers who have supported and collaborated. Martijn Kabel has made it possible to start the project in the company and Arjen Klomp has helped me for my integration to the company. David van Luijk and the Acquisition Server team have been the source of domain knowledge for me. Ton van Haren, Con Vissenberg, and the SETI team have helped me to deploy the framework on the company built environment. Ronnie Smets has helped me to test the framework in the company built environment.

I would like to express my thankfulness to my university supervisor Erik de Vink who has been guiding me on achieving the academic viewpoint of the thesis. His pertinent advice and valuable feedback has encouraged me to explore different perspectives on the project. I have obtained new vision and approach for analyzing and executing a project with help of his excellent expertise in academia.

My acknowledgment goes to everybody who has been involved in the PDEng program. Especially, Ad Aerts, Yanja Dajsuren, and Desiree van Oorschot for giving me the opportunity to be part of the program. Also, I would like to thank all coaches of the program for all the advises, reviews, feedback, and fruitful discussions. Furthermore, I would like to thank to my friends from the PDEng program for sharing wonderful experience, knowledge, culture, and many other everlasting memories.

Last but not least, my deep and sincere gratitude to my family, friends, and girlfriend for their continuous and unparalleled love, help, and support. This journey would not have been possible if not for them, and I dedicate this milestone to them.

Sodkhuu Dalaikhuu
9 September 2017

Executive Summary

Currently, the Thermo Fisher Scientific company develop their Transmission Electron Microscope software system, which is based on a component-based architecture, using the Microsoft Component Object Model (COM) technology. However, mastering COM is more time-consuming task than mastering generic programming language for developers due to the complexity of the technology. Thus, the company is moving towards separating software component implementations into layers such as COM and C++. This layered structure demands interface wrapper code which has been developed manually. The manual development of such code is inadequate in three respects:

- It is a time-consuming task which adds not as huge business value as the effort
- It is an error-prone task which includes numerous copy-paste actions from previous implementation samples
- It is an individualized task which may have different implementations depending on the developer or department

The goal of this project is to design, implement, and deploy an extensible and easy-to-use framework that generates interface wrapper code based on the information of a domain specific language. In the scope of this project, COM-to-C++ wrapper code generation based on COM Interface Definition Language (IDL) is considered as initial use-case of the framework. In order to achieve this goal, we have designed the framework architecture using the Model-Driven Architecture (MDA) which is proposed by the Object Management Group (OMG).

The framework consists of three main components such as parser, model-to-model transformer, and model-to-code transformer. The parser component parses the input IDL model to an abstract syntax tree (AST). The model-to-model transformer transforms the AST to an object-oriented interface model. The model-to-code transformer transforms the interface model to code artifacts.

We have developed a prototype, *CodeGenerationFramework*, using the architecture design and deployed it on the company built environment as a component. The component is available for all developers who wants to generate wrapper code. Also, the component is compliant with the current code structure of the company. Thus, it is easy to get and use for the developers.

The initial iterations of the prototype is verified in test components that mimics the existing code. The *Acquisition Server* has been the sample source for creating the mimicked components. The component is deployed on the company built environment as well. Therefore, it proves the framework is usable and suitable for the client company.

We have verified and validated the framework with an end-user using the existing codes for *Optics*. Several IDL files are used as input of the framework and couple of existing code artifacts are generated. However, due to time limitations and on-going code base modifications inside the company, we have barely generated actual production code artifacts and tested using the

existing unit and smoke tests for the code. We recommend that Thermo Fisher Scientific to use the framework starting from the *Energy Service Filter* interfaces. The initial concepts of the COM-to-C++ wrapper code has been generated and is ready for further modifications.

Table of Contents

Abstract	v
Foreword	vii
Preface	ix
Acknowledgments	xi
Executive Summary	xiii
Table of Contents	xv
List of Figures	xix
List of Tables	xxi
1 Introduction	1
1.1 Context	1
1.2 Preliminaries	1
1.2.1 Model-Driven Architecture	2
1.2.2 Transmission Electron Microscope	4
1.2.3 Microsoft Component Object Model	5
1.2.4 Use of COM in TEM software system	6
1.2.5 Wrapper code	7
1.3 Architectural reasoning approach	8
1.4 Outline	9
2 Problem Analysis	11
2.1 Problem statement	11
2.2 Customer objectives view	12
2.3 Application view	15
2.4 Functional view	16
2.4.1 Requirements gathering process	16
2.4.2 Non-functional requirements	17
2.4.3 Use-cases	18
3 Framework Architecture	19
3.1 Design decisions	19
3.2 Conceptual view	21
3.3 Technology choices	22
3.3.1 Domain specific language	22
3.3.2 Parser	22
3.3.3 Model-to-Model Transformer	23
3.3.4 Model-to-Code Transformer	23

4 Framework Design	25
4.1 Realization view	25
4.2 Design details	28
4.2.1 IDL grammar	28
4.2.2 Interface metamodel	30
4.2.3 Parser	31
4.2.4 Model-to-Model Transformer	31
4.2.5 Model-to-Code Transformer	35
5 Implementation	37
5.1 Implementation of the framework design	37
5.1.1 Parser	38
5.1.2 Model-to-model transformer	38
5.1.3 Model-to-code transformer	40
5.1.4 Interface grammar	42
5.1.5 Interface metamodel	44
5.1.6 Wrapper code templates	44
5.2 Custom build tool	46
6 Deployment	49
6.1 Company built environment	49
6.2 CodeGenerationFramework component	50
6.3 Demo project component	52
7 Verification and Validation	55
7.1 Verification and validation strategy	55
7.2 Test of the demo project	55
7.3 Trial with end-user	57
7.3.1 Parsing IDL files	57
7.3.2 Generating code artifacts	58
7.3.3 Feedback from the end-user	59
7.4 Proposal for COM-to-C++ wrapper code generation	59
8 Conclusions	61
8.1 Results	61
8.2 Design Opportunities Revisited	62
8.3 Future work	63
9 Project Management	65
9.1 Introduction	65
9.2 Work-Breakdown Structure (WBS)	65
9.3 Project planning	66
9.4 Feasibility analysis	67
9.4.1 Challenges	67
9.4.2 Risks	67
9.5 Project execution	68
10 Project Retrospective	71
10.1 Reflection	71
Glossary	73
10.2 Abbreviations	73
Bibliography	75

Appendices	79
A Appendix 1 Stakeholder Analysis	81
A.1 Thermo Fisher Scientific	81
A.2 Eindhoven University of Technology	83
B Appendix 2 Context Diagram	87
C Appendix 3 Functional Requirements	89
D Appendix 4 Parsed results	91
E Appendix 5 Proposal for COM-to-C++ wrapper code generation	93
E.1 COM header file	93
E.2 COM source file	94
E.3 C++ interface file	96
About the Authors	97

List of Figures

1.1	Bézivin’s view of a model-driven engineering implementation architectural style . . .	2
1.2	MDA’s suggestive metamodel transformation	3
1.3	Transmission Electron Microscope	4
1.4	TEM Software System Overview	6
1.5	Overview of simple COM type library and its wrapper code in client-server software system	7
1.6	UML class diagram of server side COM-to-C++ wrapper code	8
1.7	Overview of CAFCR model	8
2.1	Model-Based Interface Code Generation	12
2.2	Customer objectives view	13
2.3	Key-drivers diagram	14
2.4	Application view	15
2.5	Functional view	16
2.6	Use case diagram of the framework	18
3.1	Input and output artifacts of the framework in the four-layered view	19
3.2	Metamodel Transformation from DSL to interface model	20
3.3	Conceptual view	21
3.4	Conceptual diagram of the Model-Based Interface Code Generation framework . . .	22
3.5	Schematic process of the Irony parsing pipeline	23
4.1	Realization view	25
4.2	Framework design	26
4.3	Sequence diagram of code generator initialization	27
4.4	Sequence diagram of model-based interface code generation	28
4.5	Sample abstract syntax tree of IDL grammar	29
4.6	Interface metamodel design	30
4.7	Parser design	31
4.8	Model-to-Model transformer design	32
4.9	Sample AST to interface model transformation	33
4.10	Model-to-Code transformer design	36
5.1	Implementation of the framework design	37
5.2	Class diagram of the parser implementation	38
5.3	Activity diagram of Transform method	39
5.4	Activity diagram of the InitializeTemplates method	40
5.5	Activity diagram of the Generate method	41
5.6	Class diagram of the interface grammar implementation	42
5.7	Sample interface and its parse tree	43
5.8	Detailed class diagram of inteface metamodel	44
5.9	Class diagram of the wrapper code templates	45

5.10	Sample T4 template for generating C++ interface	45
5.11	Visual Studio custom build tool option	46
5.12	Visual Studio custom build tool option	47
6.1	Structure of Component Oriented Code Architecture (COCA)	49
6.2	Build flow	50
6.3	COCA structure of CodeGenerationFramework	50
6.4	Jenkins build result stored in the Holding Area	51
6.5	Jenkins build of the CodeGenerationFramework	51
6.6	COCA structure of ItfCodeGenerationDemo	52
6.7	Package diagram of the demo project component	53
6.8	Class diagram of the demo project component	54
6.9	Jenkins build result stored in the Holding Area	54
7.1	Unit test result	56
7.2	Jenkins test result	57
7.3	Parsed result of the IOptics.idl file	58
9.1	Work-breakdown structure	66
9.2	Project plan	66
B.1	Context diagram	87
D.1	Parsed result of the IImageRotation.idl file	91
D.2	Parsed result of the IOpticsDoseData.idl file	92
D.3	Parsed result of the ipeo_column.idl file	92

List of Tables

1.1	Orthogonal dimensions of model transformations with examples	3
4.1	Interface transformation rule	33
4.2	Method transformation rule	34
4.3	Parameter transformation rule	35

Chapter 1

Introduction

The Model-Based Interface Code Generation project was conducted by Sodkhuu Dalaikhuu as his Professional Doctorate in Engineering (PDEng) thesis. This chapter gives an introduction to the project by briefly revealing its context, preliminaries for background information, and architectural reasoning approach.

1.1 Context

The PDEng degree program in Software Technology is provided by the Department of Mathematics and Computer Science of Eindhoven University of Technology in the context of the 4TU.School for Technological Design, Stan Ackerman's Institute. It is a two-year, third-cycle (doctorate-level), engineering degree program designed to prepare a trainee for an industrial career as technological designer, and later on as software or system architect. The focus of the program is on strengthening technical and non-technical competencies of the trainees related to effective, elegant, and efficient design, as well as development of software in the field of Computer Science or a related field such as (but not limited to) Business Information Systems, Embedded Systems, Data Science, or Information Security. The first 15 months of the program consists of advanced training and education, including four small, industry driven training projects. During the last nine months, a major design project (PDEng thesis) in a company takes place.

Thermo Fisher Scientific (formerly known as FEI) is a world leading company that helps its customers to accelerate life science research, solve complex analytical challenges, improve patient diagnostics, and increase laboratory productivity. Its Analytical Instruments Group – Materials and Structural Analysis Division is responsible for designing, manufacturing, and supporting Transmission Electron Microscope, which provides ultra-high resolution at the sub-Ångström level (0.1 nanometer).

This thesis aims at applying the model-driven architecture concept in interface wrapper code generation for component-based software systems. The Transmission Electron Microscope is used as the implementation environment of the framework.

1.2 Preliminaries

In this section, first, the concept of Model-Driven Architecture (MDA) is explained as the main approach for realizing model-based code generation. Furthermore, the Transmission Electron Microscope is described since it is the environment of the project. Also, the Microsoft Component Object Model (COM) and its wrapper code is expounded as the use-case setup of the code generation.

1.2.1 Model-Driven Architecture

Model-Driven Architecture (MDA) is an approach, introduced by Object Management Group (OMG), for using models in software development [1]. Its main purpose is to support long-term flexibility of software in terms of implementation of new technologies, integration of new infrastructures, maintenance of the system, as well as testing and simulation. The MDA starts with the idea of separating the specification of system operation from its platform capability details. The three primary goals of MDA are portability, interoperability, and re-usability through architectural separation of concerns.

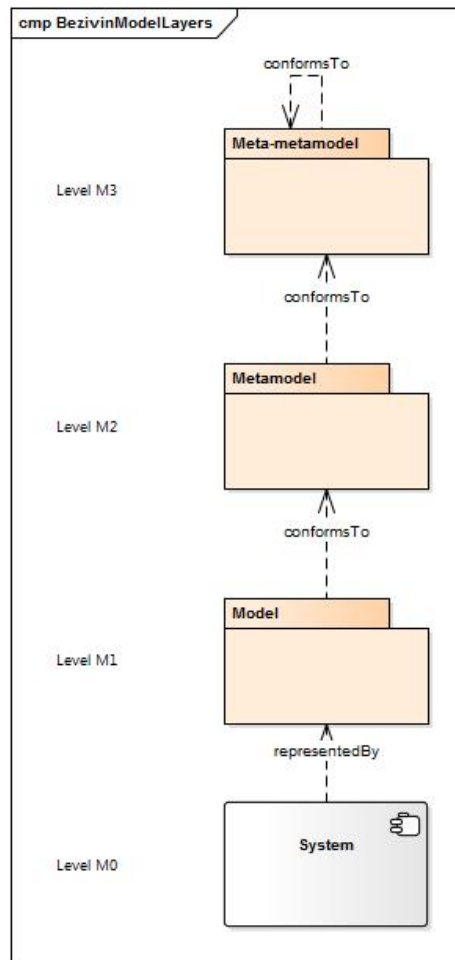


Figure 1.1: Bézin's view of a model-driven engineering implementation architectural style

The basic model driven engineering principles are proposed by Bézin in [2] and [3]. As shown in Figure 1.1, at the bottom, the level M0 is a real system. A model represents this system at level M1. This model conforms to its metamodel defined at level M2 and the metamodel itself conforms to a meta-metamodel at level M3. The meta-metamodel conforms to itself. This is very similar to the organization of programming languages like a self-representation of the Extended Backus-Naur Form (EBNF) notation, which allows defining infinity of well-formed grammars.

A particular view of a system can be captured by a model and that each model is written in the language of its metamodel. The two basic relations are called *representedBy* and *conformsTo*. The notion of a metamodel is strongly related to the notion of ontology [2]. A metamodel is a formal specification of an abstraction, usually consensual and normative. From a given system, we can extract a particular model with the help of a specific metamodel. A metamodel acts as a precisely

defined filter expressed in a given formalism.

In MDA, the meta-metamodel is called Meta Object Facility or MOF. It was developed to provide a type system, which is a set of rules that assigns a property called type to the various constructs of a computer program. The MOF is a closed and strict metamodeling architecture, which conforms to itself. Every model element on every layer is strictly in correspondence with a model element of the layer above. For defining metamodels, MOF plays exactly the role that EBNF plays for defining programming language grammars. In other words, MOF is a DSL used to define metamodels, just as EBNF is a DSL for defining grammars.

	horizontal	vertical
endogenous	<i>Refactoring</i>	<i>Formal refinement</i>
exogenous	<i>Language migration</i>	<i>Code generation</i>

Table 1.1: Orthogonal dimensions of model transformations with examples

The MDA relies on models as first class entities and it aims to develop, maintain, and evolve software by performing model transformations. Mens et al. have proposed a taxonomy of model transformation [4] to help software developers to choose a particular model transformation approach. One of the important questions that they have investigated is *What needs to be transformed into what?*. This question concerns the source and target artifacts of the model transformation, which is directly related to another important point technical space that needs to be addressed. A technical space, which is determined by a meta-metamodel (Level M3), is a model management framework containing concepts, tools, mechanisms, techniques, languages, and formalism associated to a particular technology [3]. Apart from the MDA technical space, which uses the MOF as meta-metamodel, many other technical spaces are available, including those relying on abstract syntax trees and grammars such as the EBNF. Given a model transformation, its source and target models may belong to the same or different technical spaces. In the latter case, tools and techniques to define transformations that bridge technical spaces are needed.

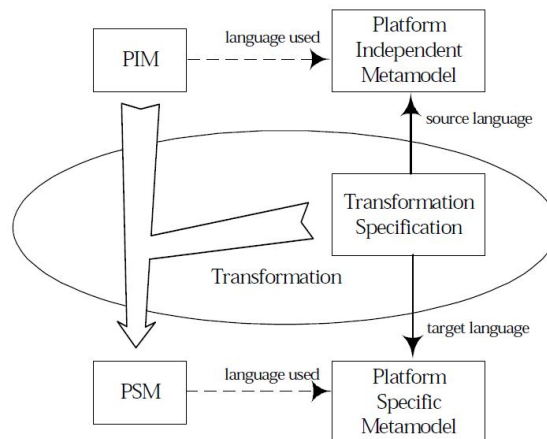


Figure 1.2: MDA's suggestive metamodel transformation

Additionally, Mens et al. have proposed to name the exogenous horizontal transformation as language migration, and the exogenous vertical transformation as code generation, as shown in Table 1.1. Language migration and code generation are the main approaches that we have used in this project. Endogenous transformations are transformations between models expressed in the same language. Exogenous transformations are transformation between models expressed using different language. A horizontal transformation is a transformation where the source and target

models reside at the same abstraction level. A vertical transformation is a transformation where the source and target models reside at different abstraction levels.

The important MDA basic concepts relevant to this project are system, model, platform independent model (PIM), platform specific model (PSM), and model transformation. Figure 3.2 shows MDA's suggestive metamodel transformation pattern. The PIM describes the system but does not show details of its use of its platform. On the other hand, the PSM combines the specifications in the PIM with the details that specify how the system uses a particular type of platform. The model transformation is the process of converting one model (e.g. PIM) to another (e.g. PSM) based on the transformation specification (or rule).

1.2.2 Transmission Electron Microscope

The Transmission Electron Microscope (TEM) is a type of microscope that shoots electrons through a specimen and then projects a magnified image onto a detector [5]. The specimen usually has to be specially prepared and held inside a vacuum chamber. TEM produces a high-resolution, gray-scale image from the interaction between the prepared specimens and electrons in the vacuum chamber. Figure 1.3 shows the Tecnai model Transmission Electron Microscope of the Thermo Fisher Scientific company.

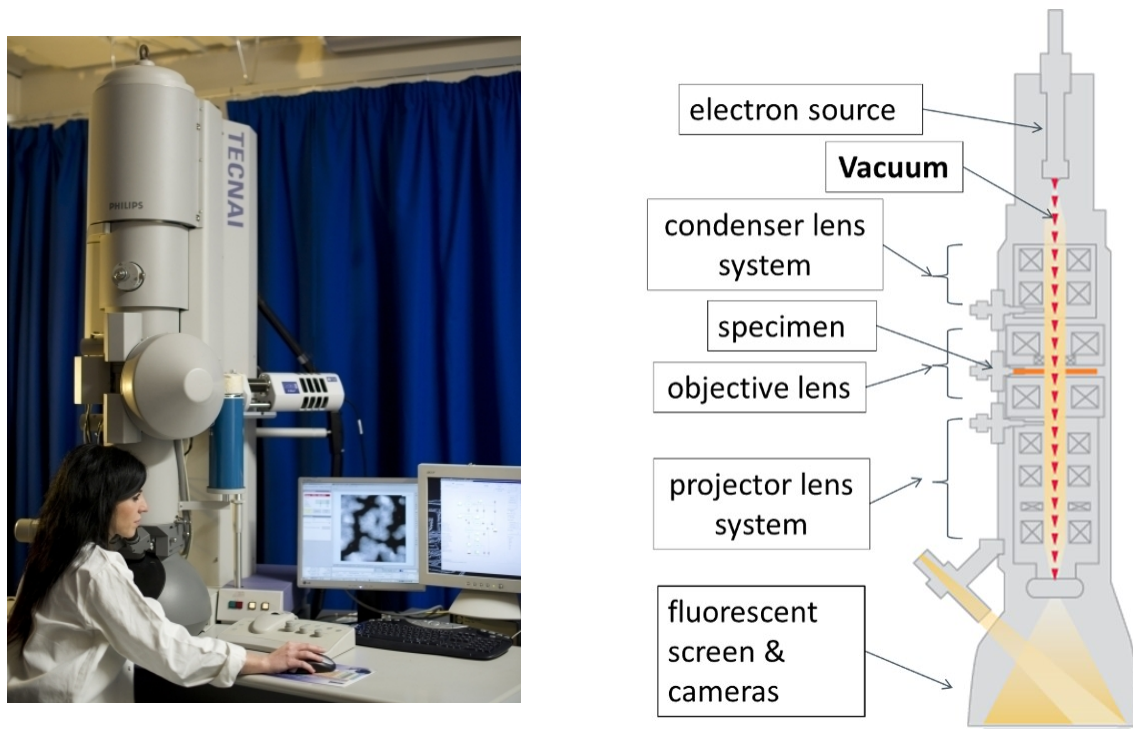


Figure 1.3: Transmission Electron Microscope

In a complex software system like the TEM, with the advancement of hardware features, the server software that connects a user application to the actual hardware component is becoming progressively intricate. As mentioned in [6], a system becomes complex in several common ways such as by definition, legacy, hardware, development location, and interdependency. They affect every phase of the software development life-cycle. The complexity of having parts that depend on inputs from other components, hardware, or systems is called complexity by interdependency. One of the well-known approaches to address this issue is the component-based architecture.

The component-based architecture's focal point is decomposition of a complex software system into particular functional or logical components that are separated using explicit interfaces containing methods and properties. One of the main benefits of this kind of architecture is increased reliability and maintainability with the reuse of existing components [7].

The software system of TEM is decomposed into components using the component-based architecture. The components use stable interfaces among themselves which makes them possible to be developed and maintained independently. Each component encapsulates a software element behavior into a reusable and self-deployable binary unit. In order to implement that, the Thermo Fisher Scientific company has used the Microsoft Component Object Model, one of the commonly used technologies.

1.2.3 Microsoft Component Object Model

Component Object Model (COM) is a Microsoft standard for creating so-called binary object that can interact in platform-independent, distributed, and object-oriented systems [8]. COM defines interfaces, which includes methods, for a software component. The implementation details, such as structure and language, of the component are left to the developer because COM is referred as a binary standard, which applies after a program has been translated to a binary machine code. In other words, the components can be implemented in different languages, and the implementations may be structurally dissimilar.

COM components are implemented in a language that can create structures of pointers and call functions through pointers. Object-oriented languages such as C++ provide programming mechanisms that can simplify the implementation of COM component. Further, COM requires that the only way to gain access to the methods of an interface is through a pointer to the interface.

The COM interaction is based on a client-server model [9]. The client application instantiates COM objects and a server produces COM services. A language independent and binary interface creation is as follows:

1. Client application instantiates a COM object using a globally unique identifier (GUID)
2. COM layer uses the registry to localize the server that can produce particular COM object
3. Server returns a pointer to the corresponding interface of a COM component to the COM layer which passes the pointer to the client application
4. Pointer returned by the COM layer to the client application points to an interface structure wherefore the COM object delivers an implementation
5. Public member functions of the interface structure are always accessible in the same manner using this interface pointer

The interfaces of a COM component is defined using the Interface Definition Language (IDL) in a ".idl" file, which contains one or more interface definitions [10]. Each interface definition is composed of an interface header and an interface body. The interface header is demarcated by square brackets. The interface body is contained in curly brackets. This is illustrated in the following example:

```
[
  /*Interface attributes go here. */
]
interface INTERFACENAME
{
  /*The interface body goes here. */
}
```

The IDL interface header specifies platform-independent attributes like globally unique identifier and version. These attributes are global to the entire interface. In other words, they apply to the interface and all of its parts.

The IDL interface body contains data types and function prototypes. The interface body can also contain attributes but they are not applicable to the entire interface. Furthermore, the interface body can contain imports, pragmas, constant declarations, and type declarations.

1.2.4 Use of COM in TEM software system

Software has an essential role in controlling and ensuring the safe operation of TEM, as well as creating a range of work-flows and applications. The TEM software system has a server software between user applications and microscope hardware. Interfaces are an important part of the software by providing access to user applications for controlling the microscope and acquiring images.

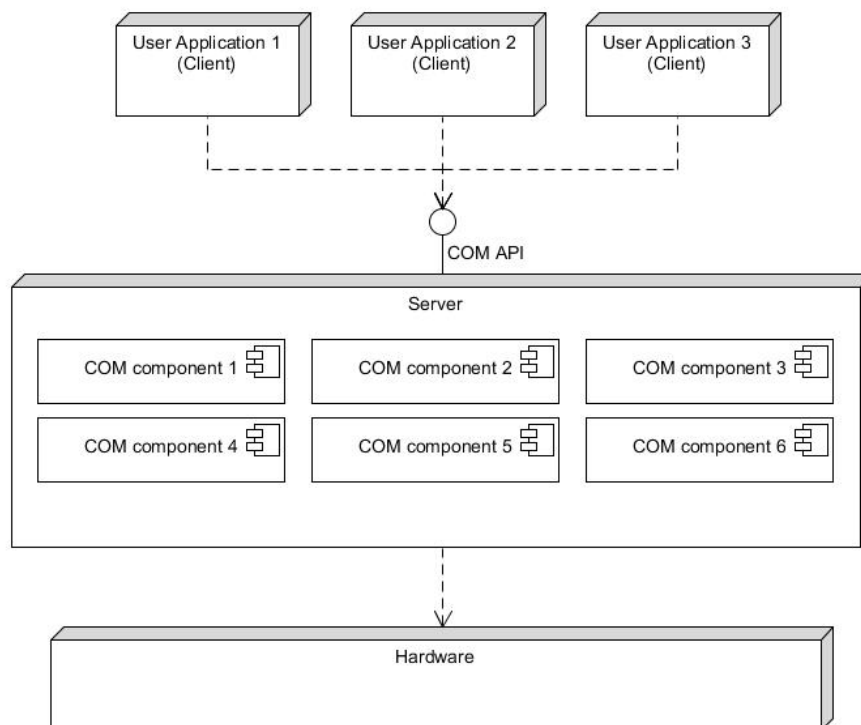


Figure 1.4: TEM Software System Overview

Figure 1.4 shows overview of TEM software system using component-based architecture and COM technology. The software of the microscope is decomposed into server and client applications. The server (Windows Service), which consists of numerous COM components, provides software interfaces to client applications for controlling the microscope hardware. The goal of the server is to abstract hardware and software differences of various microscope configurations, while maintaining relevant capabilities of subsystems. The client applications (Windows Applications) provide user-interfaces for operating the microscope. They are targeted at specific use-cases and abstracts the capabilities of the system to a convenient work-flow for the end-user. The COM API (Application Programming Interface) works between the client and server applications as a binary interface technology.

1.2.5 Wrapper code

In software engineering, wrapper code is mainly used for connecting current code to another library that has a different interface. By using wrapper code, the developer can hide low-level API code complexities and provide an equivalent interface to the high-level implementation. Usually, the wrapper class contains the same number of methods as defined in the low-level API and calls a higher level second method with no additional computation.

In the context of this project, wrapper code is considered as an adapter that allows a COM interface to be used as another interface in a generic programming language such as C++. In software engineering, the adapter pattern allows two incompatible interfaces to work together [11]. In order to do that, the adapter class contains an instance of the class it wraps and makes a call to the member method of the instance.

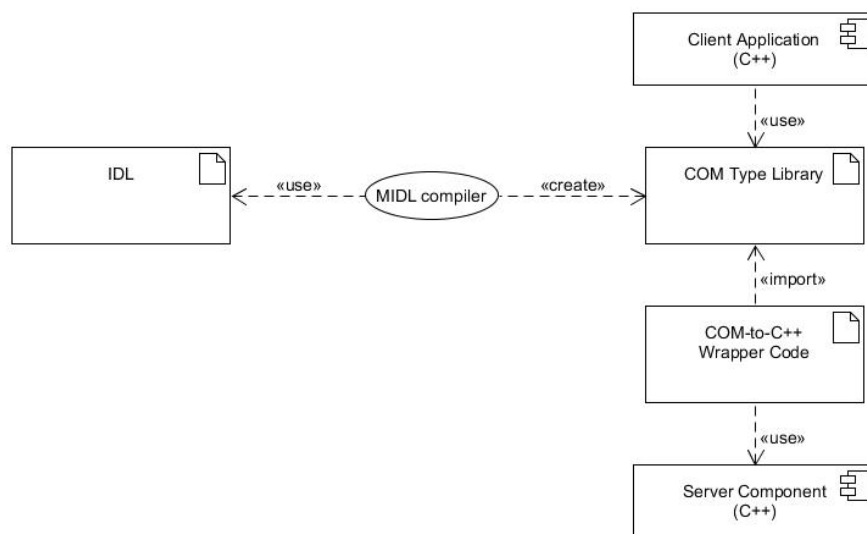


Figure 1.5: Overview of simple COM type library and its wrapper code in client-server software system

Figure 1.5 shows general structure of COM based client-server software system that we use in the scope of this project. An IDL (Interface Definition Language) file contains COM interface definitions. The file is used for creating a COM type library using the Microsoft IDL (MIDL) compiler. The COM interface is realized in the server-side wrapper code, which imports the type library and implements the interface methods. The implementation calls a higher level method of the server component.

Figure 1.6 shows simple UML diagram of the COM-to-C++ server-side wrapper in more detailed view using the object adapter pattern. The COM interface has a method. The interface is implemented by a COM class, which also has an instance of the wrapper C++ abstract class. The COM method calls the C++ abstract class method in its body. The C++ abstract class is inherited by another class, which implements the actual behavior of the method.

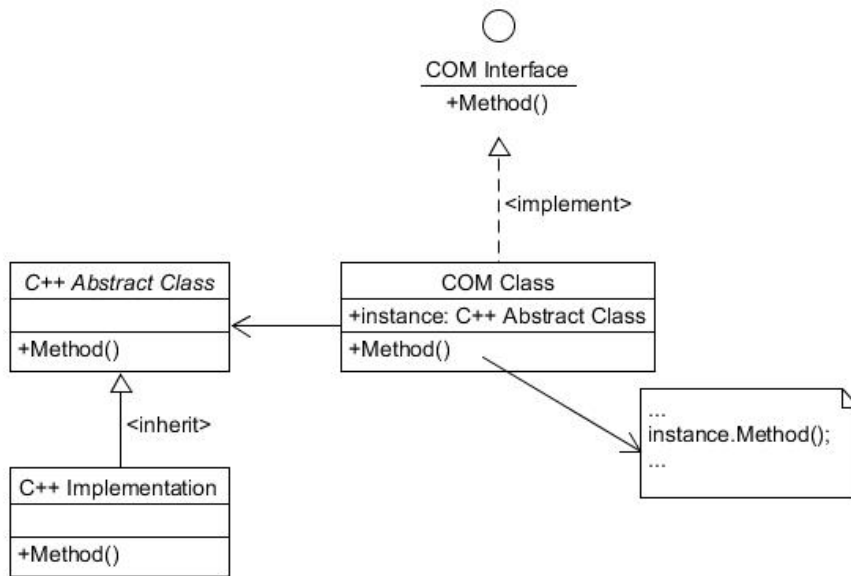


Figure 1.6: UML class diagram of server side COM-to-C++ wrapper code

1.3 Architectural reasoning approach

A system architecture is a set of decisions that gives direction to the design, implementation, and deployment phase of the system development in its environment. Its main purpose is to have design decisions consistent with the requirements of stakeholders.

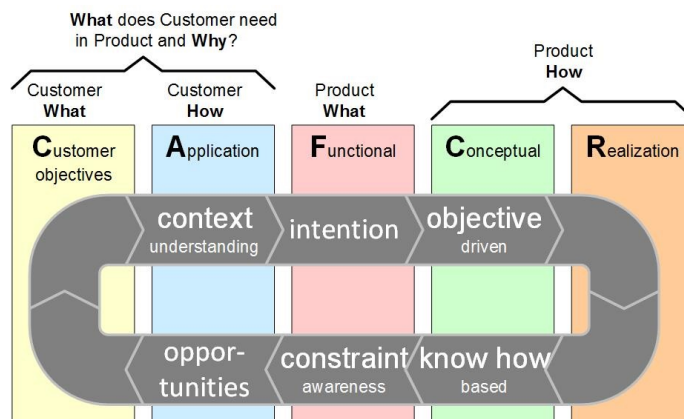


Figure 1.7: Overview of CAFCR model

The CAFCR [12] model is used for this project as an architectural reasoning approach. Comparing to the 4+1 view model, which is suitable for describing the architecture of software-intensive systems, the CAFCR is broader, more structured, and iterative approach focused on multidisciplinary embedded systems.

In CAFCR, the problem is decomposed into five views: Customer objectives, Application, Functional, Conceptual, and Realization (CAFCR). Figure 1.7 shows an overview of this approach. Per view, a collection of sub-methods is given that is filled by borrowing relevant methods from various disciplines. The customer objectives and application views compose the problem that

needs to be solved. In these views, the problem is formulated in terms of customer needs that lead to justification for the specification and design. The functional view is the junction between problem and solution. It describes preconditions for system architecture based on customer demands. The conceptual and realization view compose the technical solution that support achieving the customer objectives and application. The whole process is realized by a continuous iteration over all views.

1.4 Outline

The next chapter, Chapter 2, describes the problem that we want to solve with this project, as well as lists the non-functional and functional requirements and use-cases of the project. The requirements together with the problem analysis gives an input for the framework architecture that is elaborated in Chapter 3. The following chapters are focused on the design, implementation, and deployment phase of the project. The design of the framework is presented in Chapter 4. Chapter 5 and Chapter 6 describes the implementation and deployment phase of the project. The process of validation and verification of the framework is discussed in Chapter 7. Finally, the results and future work are addressed in Chapter 8.

The project management process during the lifetime of the project is described in Chapter 9. The last chapter of this report (Chapter 10) gives the retrospective and reflection on the project from the author's perspective.

Chapter 2

Problem Analysis

This chapter provides problem analysis by presenting essential aspects needed for the formal definition of problem statement. The first section of this chapter includes the problem statement and project goal. The last three sections address customer objectives view, application view, and functional view of the CAFCR model.

2.1 Problem statement

The software of the Transmission Electron Microscope (TEM) has numerous interfaces, reflecting the complexity of the system. As microscopes evolve, interfaces are extended and sometimes changed. Apart from the code needed to provide the actual functionality of a component, there is a significant amount of code needed for interface wrappers or technology bridges. The task to create such a wrapper code is tedious, and relatively mechanical. Also, the added business value of such task, which could be automated, is not huge as the effort.

The developers of the client company are implementing the TEM server software using the COM as the main technology for interfaces among all components. However, mastering COM is a time-consuming task, comparing to a generic programming language, due to the complexity of the technology. In order for developers to implement the behavior of a component in a generic programming language, the company uses a wrapper code such as COM-to-C++ wrapper code.

In the company, an ideal work-flow of component development would start with defining an interface with methods in the Interface Definition Language (IDL) file. Then, the developer compiles the IDL file to generate a type library (TLB) using the Microsoft IDL compiler. This TLB binary file stores information about COM object properties and methods in a form that is accessible to other applications at run-time [13]. After that, the developer implements COM-to-C++ wrapper code manually. Finally, the developer implements the component behavior in the C++ language.

Implementing a wrapper code is a repetitive, laborious, and error prone task, including numerous copy-paste actions from previous implementation samples. Normally, the wrapper code helps the developers to realize the behavioral design of the software in a generic programming language, such as C++, without concerning the COM details. However, with missing standardization and strict requirements for writing the wrapper code, the developers may be tempted to implement simple behavioral functionalities directly in the wrapper code rather than in a generic programming language. This makes the component more complex to maintain and update in the future.

The current software development paradigm, rely on the craftsmanship of skilled individuals

engaged in labor intensive manual tasks, is shifting to a new paradigm [2]. The growing pressure to reduce cost and time-to-market, as well as to improve software quality is pushing a transition to more automated approaches. The idea of software systems being composed of interconnected objects is not in opposition with the idea of the software life-cycle being viewed as a chain of model transformations. The basic principle of object technology, "Everything is an object", is moving towards the new principle, "Everything is a model".

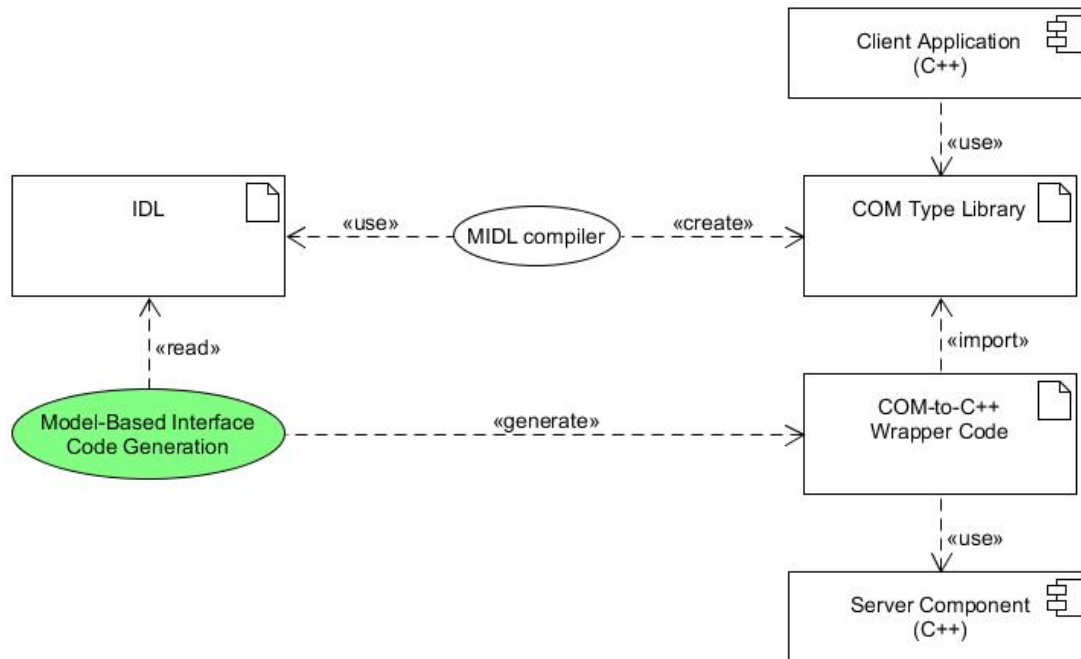


Figure 2.1: Model-Based Interface Code Generation

The goal of this project is to ease implementation and maintainability of the COM wrapper code by designing, implementing, and deploying a generic framework that generates code artifacts from domain specific models. The company's objective is to have a standard approach for COM wrapper code creation by introducing automated code generation. The research question of the project is how to apply the model-driven architecture concept for interface wrapper code generation in the context of the component-based software systems of Transmission Electron Microscope. The generation of COM-to-C++ wrapper code based on the IDL file is considered as a specific case. However, the framework should be extensible to support other domain specific languages and code artifact generations in the future. Also, the implementation has to be executed in the company development environment and has to be deployed on the company built environment.

2.2 Customer objectives view

The customer objectives view, Figure 2.2, describes *what* the customers want to gain from this project. This view is covered by a key-driver analysis that relates sharp key-drivers to a list of requirements. A key-driver can be seen as the primary goal of the customer. These goals are refined by application-drivers into more elaborate definitions. Finally, a set of requirements is derived from the application-drivers. These requirements express what are the achievements and limitations of the product. The detailed stakeholder analysis is included in Appendix A.

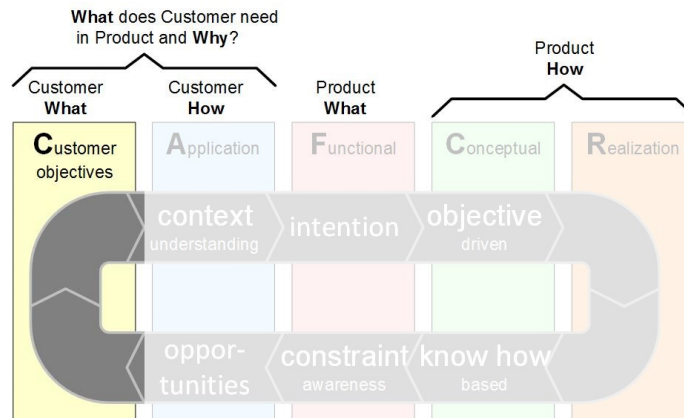


Figure 2.2: Customer objectives view

A good understanding of the customer and the product requirements can be obtained from the key-driver's diagram, shown in Figure 2.3. The customers of this project are primarily concerned with minimizing the effort of creating wrapper code by automatically generating the code using a framework. This main key-driver is decomposed into concrete ones by classifying in terms of ease-of-use, extensibility, and reliability. These three concrete key-drivers are refined by detailed application-drivers that finally lead to requirements. As same as the other views of the CAFCR model, the key-driver's diagram is improved iteratively, based on the other views, results, and achievements, during the entire project duration.

The first concrete key-driver is the ease-of-use of the framework. The framework should be easy to use for interface developers without requiring a steep learning curve. After discussing with the company stakeholders, we have evaluated that deploying the framework on the company built environment using their current technologies and tools would be suitable for this purpose. Therefore, this concrete key-driver is refined by an application-driver that deploy the product on the company built environment. This application-driver has derived four requirements such as to use client company's current tools and environment for the framework development, to use a DSL file as the input of the framework, to generate wrapper code as the output of the framework, and to create a custom build tool for the DSL file. The custom build tool would combine the client company's current compiler with our code generator framework.

The second concrete key-driver is the extensibility of the framework. The framework should be easy to extend for future code generation projects such as test generation and document generation. This concrete key-driver is refined by two application-drivers: support other domain specific languages as well as generate other types of text artifacts. The first application-driver has derived two requirements such as to use an extensible and replaceable tool for DSL parser as well as to implement the parser as a component of the framework, which could be updated or replaced easily in the future. The second application-driver has derived one requirement that is to use an extensible and replaceable tool/technology for text artifact generation.

The third concrete key-driver is the reliability of the framework. The framework should be validated and verified. This concrete key-driver is refined by two application-drivers that are to verify the generated code artifacts, and to validate the framework with company stakeholders. The first application-driver has derived one requirements that is to use existing test environment for verifying the generated wrapper code artifacts. The second application-driver has derived one requirement that is to perform a trial run with an end-user.

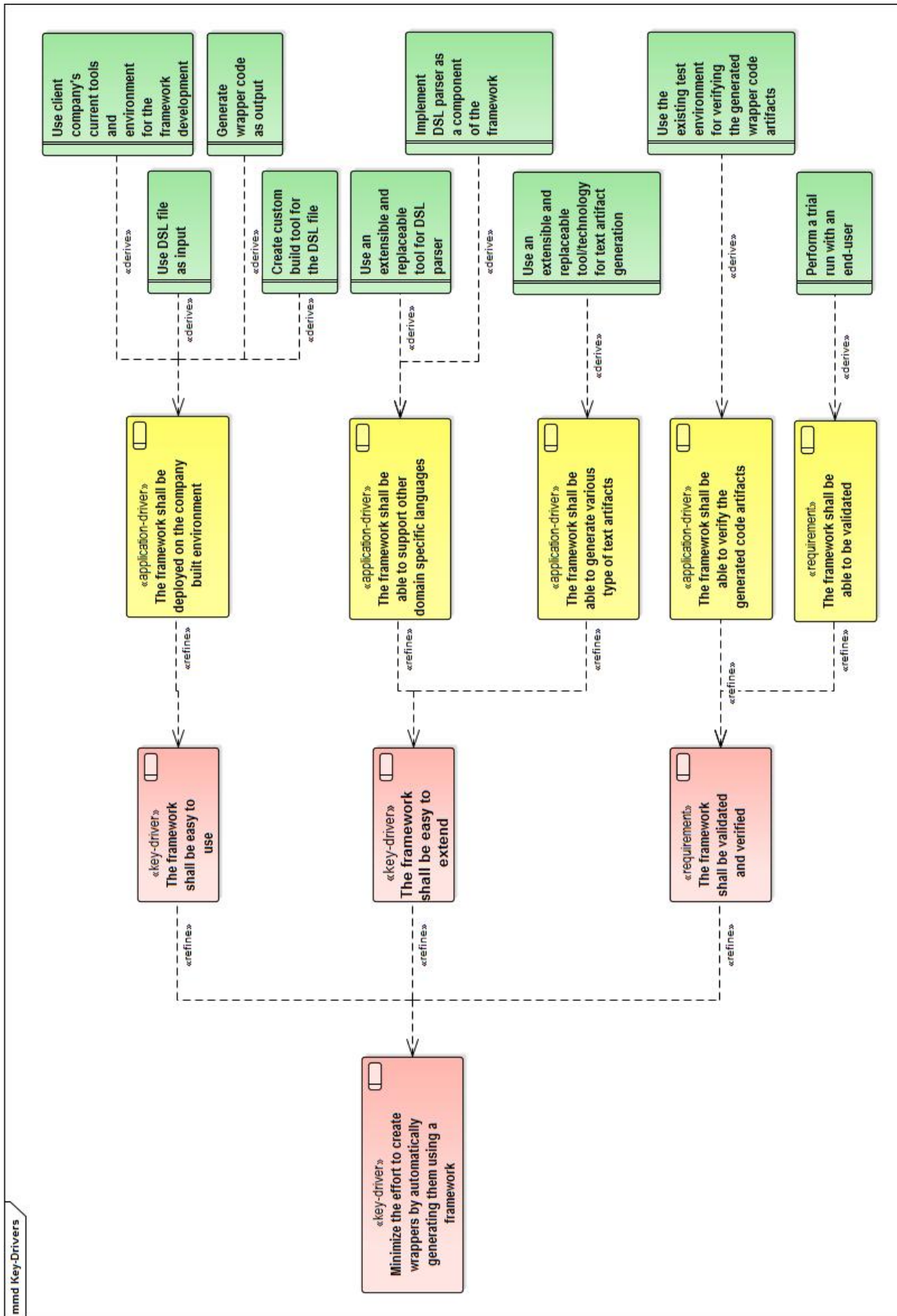


Figure 2.3: Key-drivers diagram

2.3 Application view

The application view, Figure 2.4, describes *how* the product is placed in its environment. This view is covered by context analysis that includes related work, relevant technologies, and deployment environment.

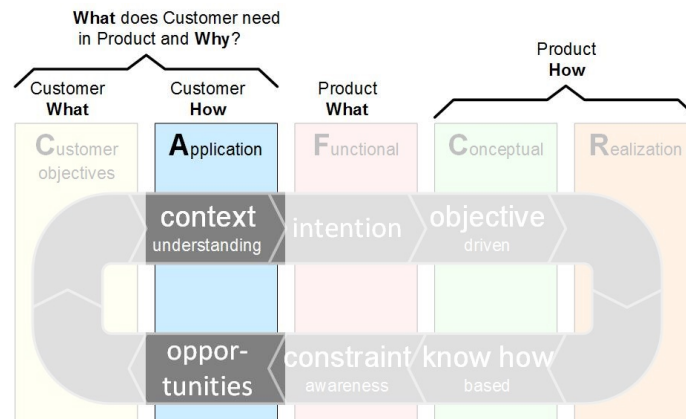


Figure 2.4: Application view

The context analysis includes technologies and projects that are relevant to the framework. It has three main items: company built environment, Auto Component Interface project, and Model-Based Interface Framework project. The company built environment includes technologies, tools, and code bases relevant to the current project. The two projects are previously developed in-house prototypes of the client company. They provide proof-of-concept for the framework design decisions and technology choices. The context diagram is included in Appendix B.

The first item of the context analysis is the company built environment. The TEM software code base is based on a component-based architecture, which enables the developers to implement and integrate components independently. Furthermore, the code base includes all wrapper codes for components. COM IDL is the main technology that the company use for realizing the component-based architecture. Visual Studio is the primary tool that they use for software development. The company built environment is elaborated in Chapter 6.

Another important item of the context analysis is the Auto Component Interface project. The Thermo Fisher Scientific company has been introducing automatic code generation in the microscope software development for the past few years. In particular, there are two previous works that are relevant to this thesis and the Auto Component Interface project is one of them. It is aimed at generating C++ interface components based on a domain specific language called FIDL (FEI Interface Definition Language). This project is a proof-of-concept for generating components using the combination of Irony .NET Language Implementation Kit and Microsoft T4 technologies. The Irony is used for implementing the parser of the framework. The Microsoft T4 (Text Template Transformation Toolkit) is used for generating the output code artifacts based on the outcome of the parser. However, the project does not support the company's current standard interface technology, Microsoft COM, as well as the project is a prototype, which does not fulfill the production-quality.

The second project, Model-Based Interface Framework (MBIF) [6], is another in-house project of the company, which is a previous PDEng thesis. The project is aimed at improving interface quality by creating a generic framework for test, wrapper code, and document generation. The project was developed in the Eclipse xtext/xtend framework. This technology

decision was made with analysis of existing approaches in seven criteria: customization of language/model definition, customization of artifact generation, simplicity for model definition, maturity, extensibility, licensing cost, and cost of usage/integration. Based on these criteria, they have decided to go with a domain specific language (DSL) based approach by defining their own language. The project is a proof-of-concept for generating test, simple wrappers, and documents using a DSL. However, the framework is not directly deployable on the company built environment due to technology differences. In particular, the company uses Visual Studio tool for platform development, which means the interface developers would need to learn the new tool in Eclipse environment. Also, updating the tool when new functionalities or features are introduced is a complicated task, especially, when the tool is not stable. Furthermore, this project was mainly focused on the test code generation so that, apart from sample case of a simple wrapper code generation based on its own DSL, the production-quality COM-to-C++ wrapper code generation has been left for future work.

The company stakeholders have three ideas, based on the aforementioned pilot project experiences, for the future of automated code generation in the software development of the company. First, extend the Auto Component Interface project by developing additional features, and improve the project to production-quality considering the suitability with the company built environment. Second, extend the MBIF project to generate mature and verified wrapper code artifacts but this project would not be directly suitable with the company built environment due to the technology differences. Hence, it would require a steep-learning curve for interface developers to use it in the production. Third, create a generic framework that supports both concepts in the company built environment. The latter is more attractive to the clients because it allows future code generation projects as an extension and it would be usable for the interface developers. We have decided to choose the last option for the scope of this project.

2.4 Functional view

The functional view describes *the what* of the system. This view is covered with requirements analysis and use-case analysis. The following subsections describe non-functional requirements and use-cases of the framework. The list of functional requirement is included in Appendix C.

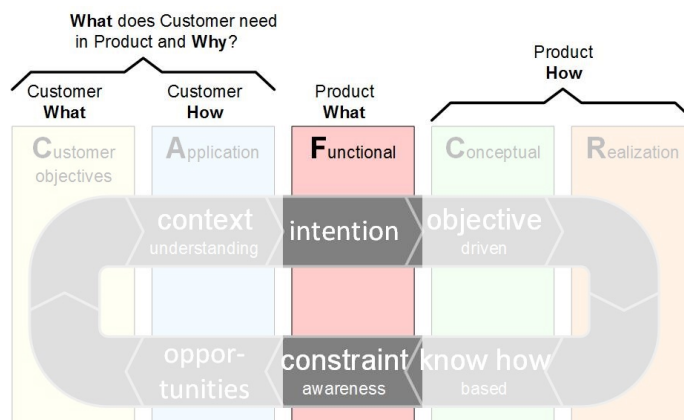


Figure 2.5: Functional view

2.4.1 Requirements gathering process

The requirements gathering process is occurred in parallel with the development activities. There are two phases of requirement definition in the process, high-level and detailed. The high-level

phase starts at the beginning of the project and lasts no longer than the third month. This stage focuses on defining goals and general requirements without detailed specifications. Decisions according to technologies and expected artifacts are the main outcome. The detailed phase is centered on clear and specific definitions of the previously established high-level requirements. The requirements are defined in an agile style which results in the creation of options and recommendations.

Two sets of requirements are identified, non-functional and functional. Each functional requirement has a priority selected from three levels, which are *must*, *should*, and *optional*. The *must* level is an absolute requirement that has to be fulfilled. The *should* level is a requirement that could have valid reasons in particular circumstances to ignore. However, the full implications must be understood and carefully weighed before choosing a different course. The *optional* level is a truly extra requirement that is not mandatory but would be preferable if it fits in the project time-frame.

2.4.2 Non-functional requirements

Extensibility

The non-functional requirement of extensibility is described in two subsections: coupling and inserting/replacing. The coupling is about how easy to separate parts of the framework. The inserting/replacing is about how much effort is required to insert/replace/modify an existing component/part of the framework.

The measurement of coupling is distinguished in three metrics: easy, reasonable, and difficult. The framework is easy in terms of coupling if parts are separate in packages and components that allow a visual and logical separation. The coupling is reasonable if some sections are still coupled because of implementation reasons. The coupling is difficult if many dependencies and cross-referenced elements.

The measurement of inserting/replacing is distinguished in three metrics: easy, reasonable, and difficult. The framework is easy if modifications involve a small number of changes in the full structure. The framework is reasonable if modifying involves changing structures, classes, in other components. The framework is difficult if changes in all the structure are required to support modifications.

Ease of use

The non-functional requirement of ease-of-use is described in three subsections: time required for learning the tool, time required for creating a wrapper code, and intuitiveness. The time required for learning the tool is about how long does it take to get used to the tool. The time required for creating a wrapper code is about how long does it take to learn the process of generating a wrapper code. The intuitiveness is about how easy is it to use the tool and associate the tool concepts with the domain concepts.

Each subsections have metrics. The measurement of the first two subsections are in time metric, which is defined based on intuition and experiences. The measurement of the intuitiveness is distinguished in three metrics: high, medium, and low. The intuitiveness is high if the framework is easy to associate concepts and to use them directly. The intuitiveness is medium if the framework is easy to associate the concepts, but usage of them is not completely clear. The intuitiveness is low if the concepts are not clear enough to associate and use them.

Reliability

The non-functional requirement of reliability is considered as verification activity in our project. The measurement of the verification is defined as the existing unit and smoke test environment of the client company. The generated code artifacts should be reliable in terms of functionality. The code should have the same functionality as the manually developed code. Hence, the code should be verified by the existing tests.

2.4.3 Use-cases

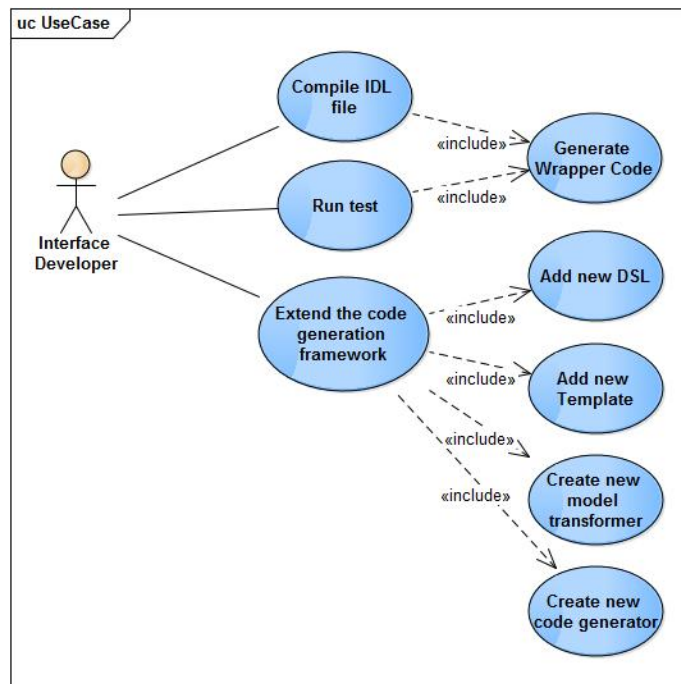


Figure 2.6: Use case diagram of the framework

The main use-case of the framework is generate wrapper code. This use-case is included in two other use-cases such as compile IDL file, and run test. As mentioned in the key-drivers diagram (see 2.3), the framework should be easy to use for interface developers. In order to fulfill this application-driver, we have decided to integrate the wrapper code generation action with the action of compiling an IDL file. Another use-case that includes wrapper code generation is the run test use-case. The framework would need to generate wrapper code, when interface developer runs a test, if the IDL file is modified after the last run of the test.

The other important use-case is extend the code generation framework. This use-case includes four other use-cases such as add new DSL, add new template, create new model transformer, and create new code generator. As mentioned in the key-drivers diagram as well, the framework should be easy to extend for other DSL's and other code artifact generations. Therefore, it should be easy to add new DSL's and code templates. There might be new model transformer and code generator in the framework depending on the new DSL. Hence, the framework should be easy to add them as well.

Chapter 3

Framework Architecture

Architecture designing is a matter of defining a structured solution, which meets the project preconditions, for the problem. This chapter covers from relevant decisions and reasoning to high-level structure of the framework.

3.1 Design decisions

The design decisions of the framework architecture has started with defining the input and output artifacts because these artifacts are the definite use-case of the project. Based on the project requirements, the input of the framework is a domain specific language (DSL) model (e.g. IDL model), which is considered as a platform independent model (PIM), and the output is a code artifact (e.g. COM-to-C++ wrapper code), which is considered as a platform specific model (PSM).

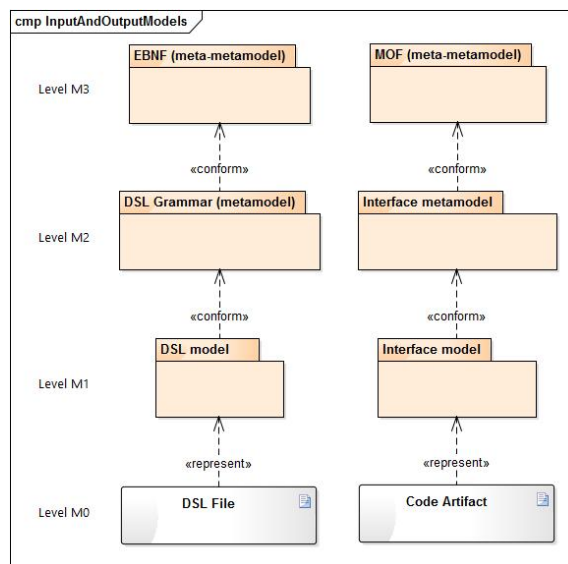


Figure 3.1: Input and output artifacts of the framework in the four-layered view

The input and output of the framework are in two different technical spaces, which is shown in Figure 3.1 using the four-layered view. The first technical space is the Extended Backus-Naur Form (EBNF), which rely on abstract syntax trees (AST) and grammars. As same as we have mentioned in the introduction chapter, the input DSL file contains model definitions. These models are represented by DSL models, which conform to the corresponding grammar in the metamodel layer. In the scope of our project, we consider all DSL grammars conform to the EBNF

because most programming language standards use some variant of EBNF to define the grammar of the language [14]. The second technical space is the Meta Object Facility (MOF). The output code artifacts are represented by interface models, which conforms to an interface metamodel. We consider the output code artifacts are structured in an object-oriented programming language. Therefore, the interface metamodels conform to the MOF.

Figure 3.2 shows a transformation from DSL model to interface model in the form of Object Management Group (OMG)'s suggestive metamodel transformation pattern [3]. The transformation rule defines a meta-level direct mapping between metamodels, which is used for the concrete transformation between the DSL and interface models.

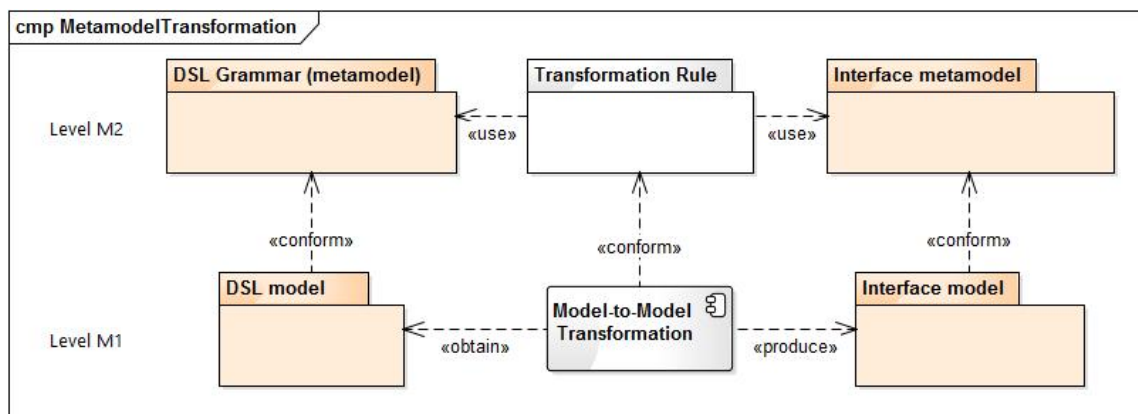


Figure 3.2: Metamodel Transformation from DSL to interface model

As shown in Figure 3.2, a model transformation is the corner stone of this project. Model transformation approaches can be classified in two major categories as proposed in [15]: model-to-model and model-to-code. A model-to-model transformation translates between source and target models whereas a model-to-code transformation translates between source model and target programming language.

The model-to-model transformations are useful when intermediate models are needed for connecting a large abstraction gap between PIM and PSM [15]. In our case, code artifact generation from DSL file is considered as a large abstraction gap because they are represented in different technical spaces. Hence, the model-to-model transformation (or language migration) is used as a preprocessing step for the model-to-code transformation (or code generation) in our framework.

In order to transform a DSL model to an interface model, a concrete data structure technique is needed. Here, an abstract syntax tree (AST) is used as an intermediate model, which represents the DSL model information in a tree-like structure. For transformation between the intermediate model and the interface model, structure-driven approach is considered as pragmatic during literature review and practical implementation. This kind of approach has two distinct phases [15]: creating the hierarchical structure of the target model, and setting attributes and references in the target. In other words, the main idea is mapping source model informations to target model elements. The mapping is performed between AST models and interface models conforming to a transformation rule between the respective metamodels.

3.2 Conceptual view

The conceptual view describes *how* the product is depicted at a high-level. This view is covered by conceptual diagram based on the design decisions.

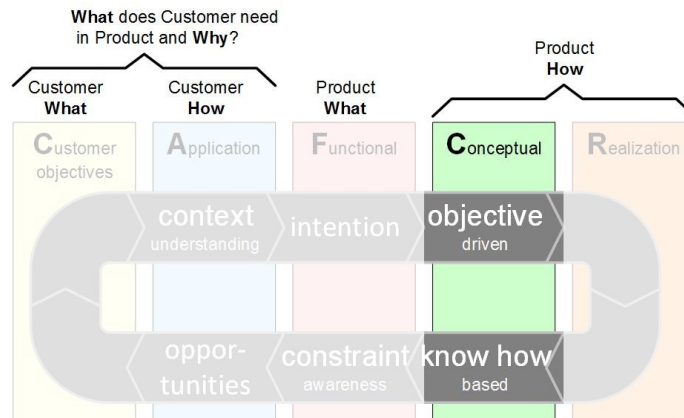


Figure 3.3: Conceptual view

Three main steps have arisen during the design decision process for generating code artifact from DSL file. The first step is to parse an input model to an AST. The second step is to transform the AST to an interface model. The third and final step is to transform the interface model to a code artifacts.

Figure 3.4 shows the conceptual diagram of our code generation framework. The aforementioned three steps are represented by three components in the framework: Parser, Model-to-Model Transformer, and Model-to-Code Transformer. The parser reads a DSL file and parses its definitions, using the corresponding DSL grammar, to an AST. The model-to-model transformer gets an AST as input and uses a transformation rule, which uses corresponding DSL grammar and interface metamodel, to transform the tree-like information structure to an interface model. The model-to-code transformer gets the interface model from the model-to-model transformer and uses templates to generate code artifacts.

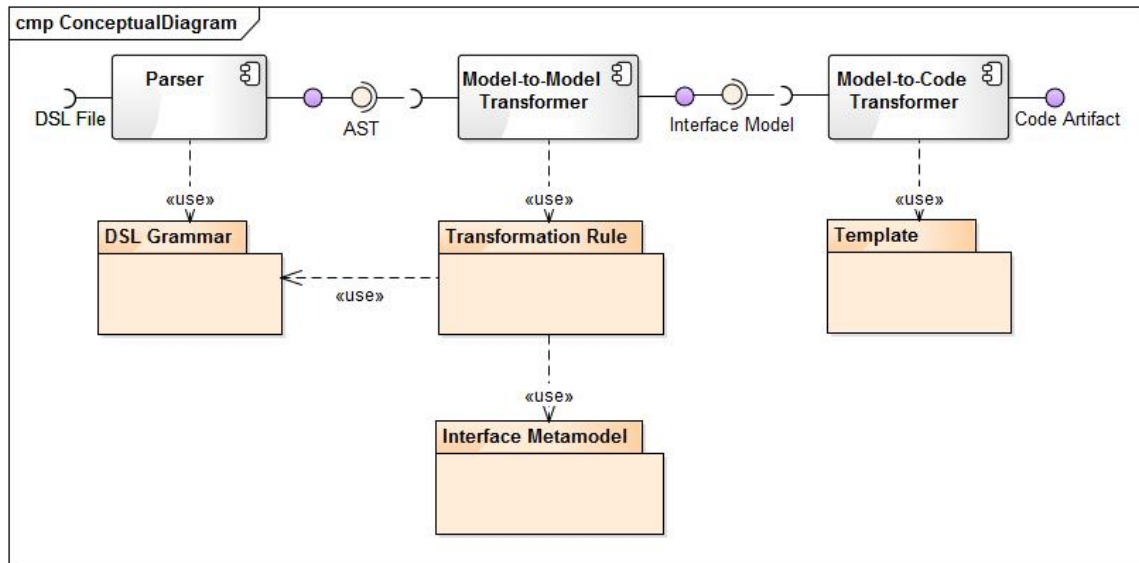


Figure 3.4: Conceptual diagram of the Model-Based Interface Code Generation framework

3.3 Technology choices

3.3.1 Domain specific language

In the scope of this project, the Interface Definition Language (IDL) is used as DSL. The IDL is a specification language for COM interface modeling in a simple textual notation. It describes structural part of the COM interface model. The syntax of the IDL is defined in the IDL grammar which conforms to the Extended Backus-Naur Form (EBNF). The design of the IDL grammar is introduced in Chapter 4.2.1.

3.3.2 Parser

The Irony .Net Language Implementation Kit [16] is used as the parsing technology of the framework by following the proof-of-concept of the Auto Component Interface project mentioned in Chapter 2.3. It is an open source, C# and .NET based technology for constructing a parser or compiler. Furthermore, comparing to similar technologies like Eclipse xtext/xtend, it is directly suitable for the software development environment of the client company. Hence, this technology fulfills the necessity of having an usable tool for interface developers.

Parsing technology is used in many modern applications such as compiler development, script engines and expression evaluators, source code analysis and refactoring tools, template based code generators, formatters and colorers, data import and conversion tools, and more [17]. Reliable and straightforward method of implementing a parser is important for developers. Therefore, the Irony .NET Language Implementation Kit is developed by Roman Ivantsov [16]. Irony is an attempt to fix the situation that parser construction technology did not change much since mid-seventies. It aims to bring the power of a modern language environment like C# and .NET into the parser and compiler construction field.

The process of the Irony parser is shown in Figure 3.5. In Irony, the standard processing pipeline of a parser is extended by adding optional processing modules called Token Filters between the scanner and the parser. The scanner module eliminates whitespace and comments, and groups input characters into meaningful words represented by objects called tokens such as numbers, string literals, keywords, variables, and special symbols. The token filters are special processors that intercept the token stream between the scanner and the parser. They remove from

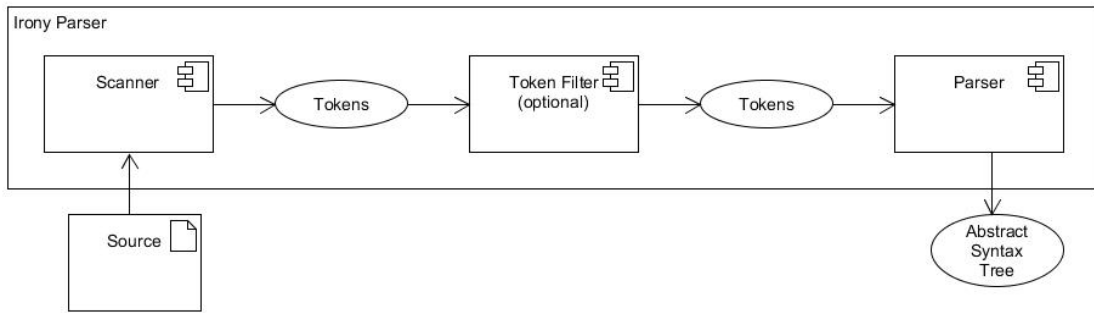


Figure 3.5: Schematic process of the Irony parsing pipeline

or add tokens to the original stream. The parser module is based on the so-called LALR parser. It is an abstract machine with a finite number of internal state (or Deterministic Finite Automation) which starts in initial state and moves from one state to another while consuming input tokens and executing predefined actions until it arrives at final state. The final result of the parsing process is an abstract syntax tree.

3.3.3 Model-to-Model Transformer

There are various model transformation tools and technologies such as ATL, JTL, ETL, Kermeta, QVT, Atom, Acceleo, xtext/xtend, xpanse, JET, and MOFScript. However, most of them are implemented in Eclipse or Epsilon platform and none of them supports .NET framework [18]. Therefore, in order to fulfill the ease-of-use and extensibility requirements, we have decided to implement the model-to-model transformer component in C# by using direct mapping approach from source to target model. The design and implementation of this component is introduced in the following chapters.

3.3.4 Model-to-Code Transformer

One of the important requirement of the framework is to use an extensible and replaceable tool/technology for text artifact generation. Furthermore, in order to fulfill the non-functional requirement of ease-of-use, the choice of having a human-readable definition of the output artifact is needed. Hence, template-based approach is used for generating code artifacts from interface model. This type of approaches are available in the majority of MDA tools. A template usually includes a target text that access the source model information and perform code selection and iterative expansion.

The Microsoft Text Template Transformation Tool (or Microsoft T4), one of the mature tools for the template-based approach, is used as the code generation technology in our framework. Microsoft T4 has two type of templates [19]: run-time and design-time. The run-time text templates are suitable for our framework because this type of templates are executed in the application to produce text strings, typically as part of its output, whereas, the design-time templates does not work with the application execution, only generate code artifacts in design-time. Furthermore, the developer can define as many templates as needed using the input interface model information, which enables multiple target model generation from single source model.

Chapter 4

Framework Design

The previous chapter introduces the framework architecture based on design decisions. In this chapter, we introduce the design of each element of the architecture in more details. The first part of this chapter introduces general design of the framework. The second part explains each design elements in more details, yet the implementation level details are not included.

4.1 Realization view

The realization view describes *how* the product is designed in low-level. This view is covered by class and sequence diagrams of the framework design. As illustrated in previous chapter, see Figure 3.4, the framework consists of three main modules: Parser, Model-to-Model Transformer, and Model-to-Code Transformer. This section expounds the generic design of the three main modules.

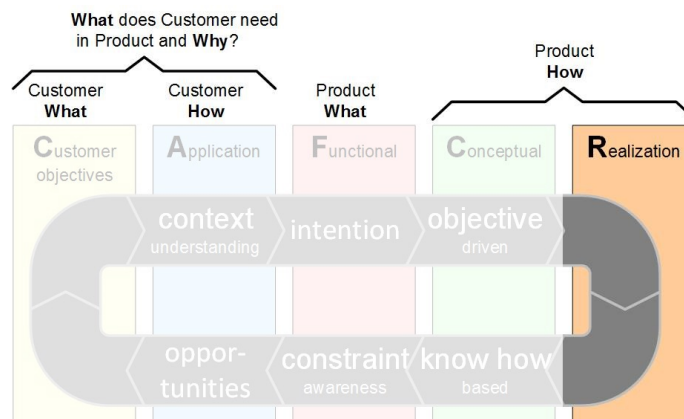


Figure 4.1: Realization view

The framework design, which is shown in Figure 4.2, is the direct realization of the conceptual diagram that we have introduced in the previous chapter. The abstract factory design pattern is used in the framework to encapsulate grammar, model-to-model transformer, and model-to-code transformer factories without specifying their concrete classes. This pattern separates the details of implementation of a set of objects from their general usage and relies on object composition, as object creation is implemented in methods exposed in the factory interface. The parser of the framework is an extension of the Irony parser.

The client *ModelBasedInterfaceCodeGenerator* class creates a concrete implementation of the abstract *CodeGeneratorFactory* and uses the generic interface of the factory to create the concrete

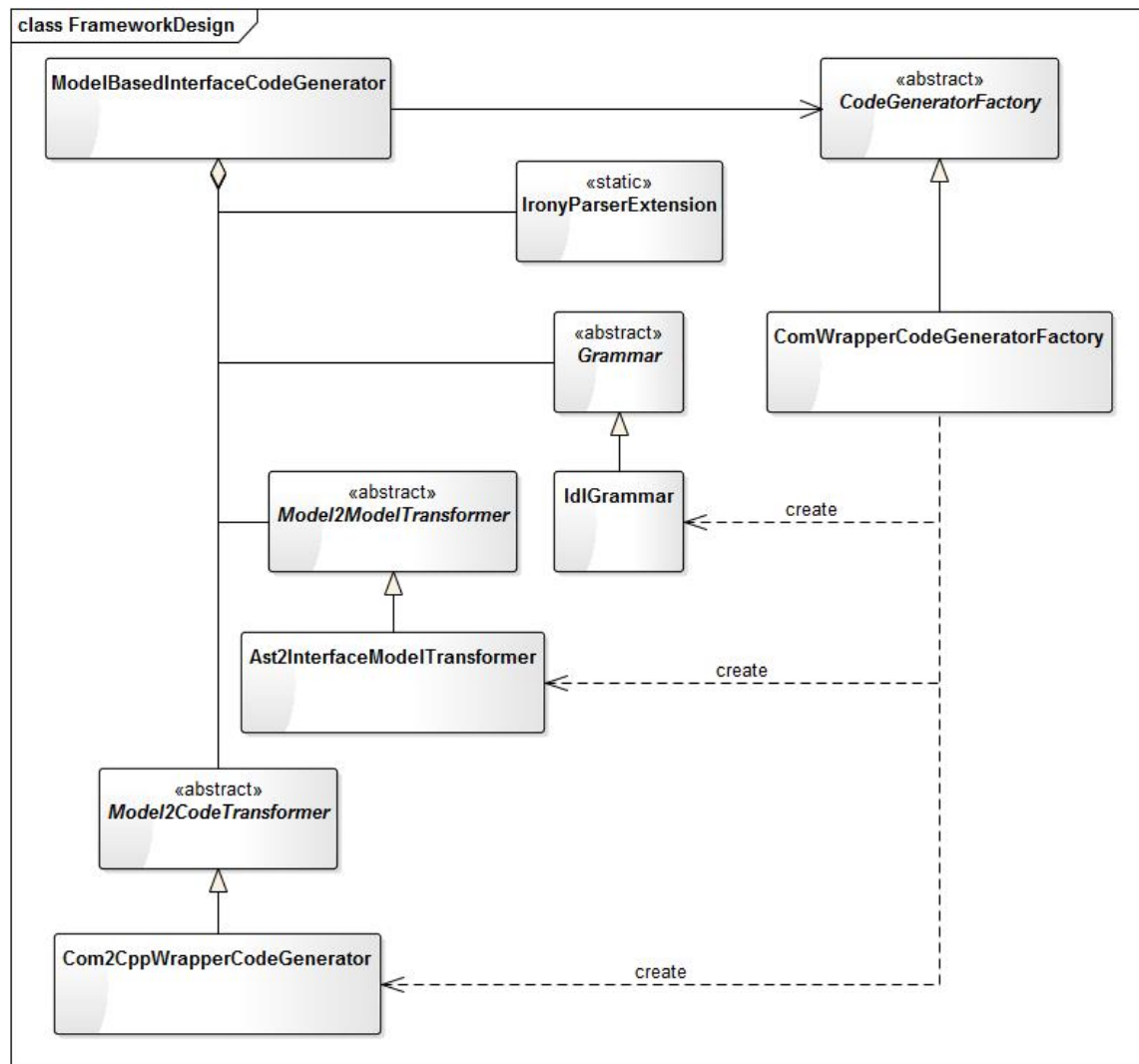


Figure 4.2: Framework design

objects of each components of the architecture. The client doesn't know which concrete objects it gets from each of these internal factories, since it uses only the generic interfaces of their components.

Figure 4.3 shows sequence diagram of code generator initialization using the factory method pattern. The *ModelBasedInterfaceCodeGenerator* class calls three object creator methods from *CodeGeneratorFactory* such as *CreateModel2ModelTransformer*, *CreateCodeGenerator*, and *GetGrammar*. Each methods, implemented in concrete class *ComWrapperCodeGeneratorFactory*, create new objects of concrete model transformer, code generator, and grammar respectively using *Ast2InterfaceModelTransformer*, *Com2CppWrapperCodeGenerator*, and *IdlGrammar* concrete classes.

Figure 4.4 shows generic sequence diagram of the model-based interface code generation framework. The three main steps that we have introduced in the architecture is represented here as method calls from the corresponding classes. First, *ModelBasedInterfaceCodeGenerator* class calls *Parse* method from *IronyParserExtension* using input file path and grammar as parameters. The call returns a parse tree (AST) which represents models in the file conforming to the grammar. Second, the class calls *Transform* method from *Model2ModelTransformer* using the parse tree as parameter.

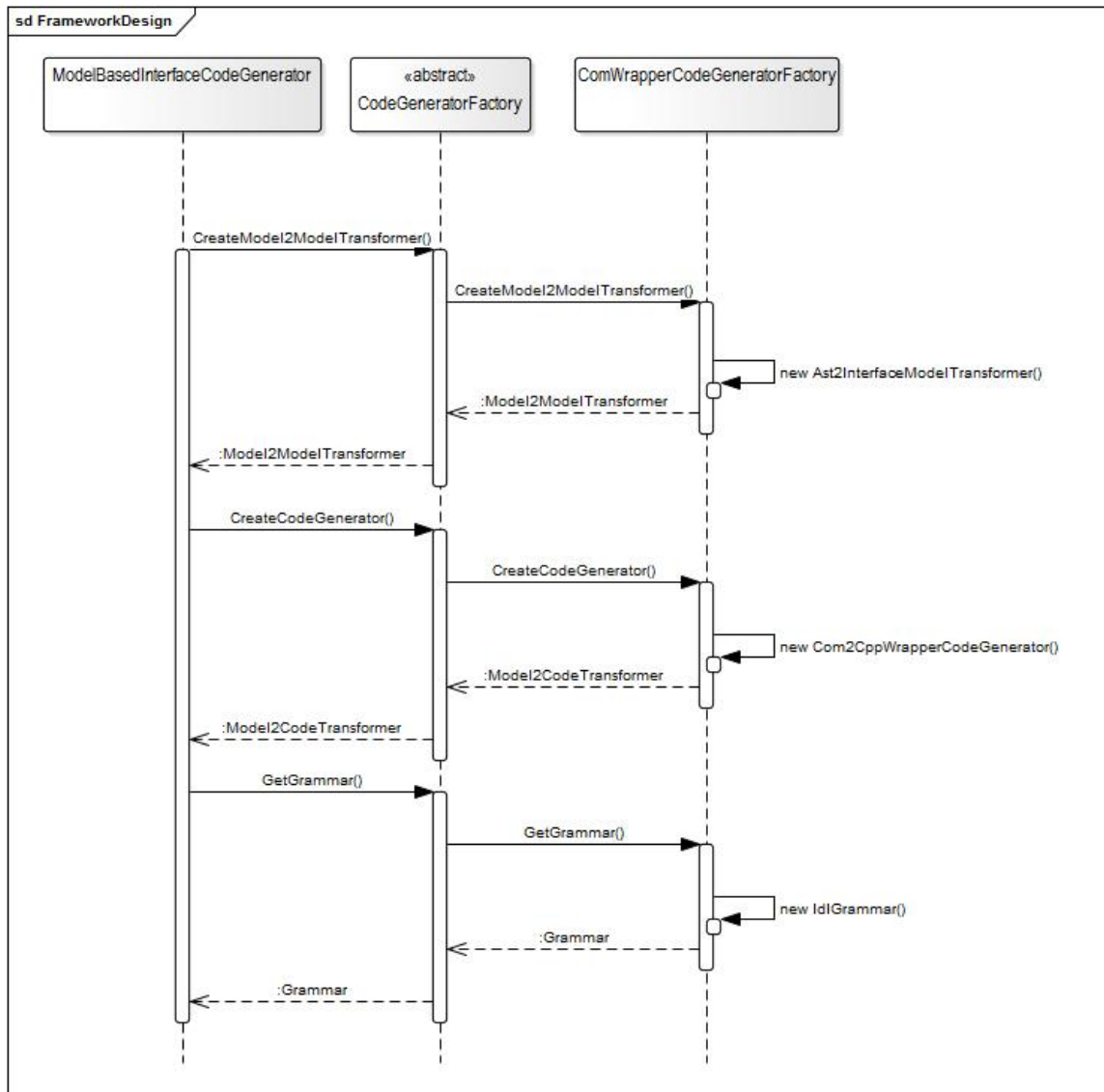


Figure 4.3: Sequence diagram of code generator initialization

The call returns a namespace that contains interface models in object-oriented structure. Third, the class initializes output code artifact templates by calling *InitializeTemplates* method from *Model2CodeTransformer* using the namespace and input file path. Finally, the class calls *Generate* method from *Model2CodeTransformer* to generate code artifacts.

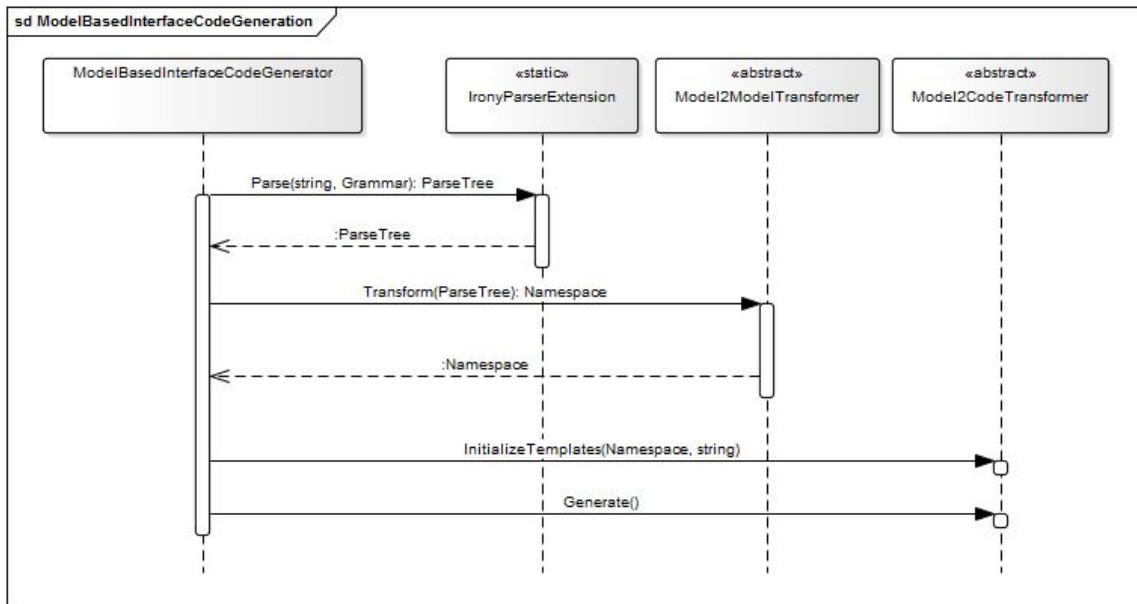


Figure 4.4: Sequence diagram of model-based interface code generation

4.2 Design details

4.2.1 IDL grammar

A grammar describes the actual structure of language that is defined as a set of rules. These rules are based on the formal language theory, in which a grammar is a set of production rules for strings in a formal language. These rules declare how to form strings from the language's alphabet that are valid according to the language's syntax. A grammar does not describe the meaning of the strings or what can be done with them in whatever context, only their own.

Our IDL grammar design was inspired by OMG's IDL grammar [20]. OMG IDL is a language that allows unambiguously specifying interfaces that client objects may use and object implementations provide, as well as all needed related constructs such as exceptions or data types. Data types are needed to specify parameters and return value of interface methods. They can be used also as first class constructs. Sample interface building block of the IDL grammar is as follows:

```

<specification> ::= <definition>+
<definition> ::=+ <interface_dcl> ';'
<interface_dcl> ::= <interface_def>
<interface_def> ::= <interface_header> [ <interface_inheritance_spec> ] "{" <
  interface_body> "}"
<interface_header> ::= "interface" <identifier>
<interface_inheritance_spec> ::= ":" <interface_name> { "," <interface_name> }*
<interface_name> ::= <scoped_name>
<interface_body> ::= <member>*
<member> ::= <method_dcl> ";"
...
...
...
  
```

The building block gathers all the rules needed to define basic interfaces. Several symbols are used above as EBNF notation. The explanation of each symbols are as follows:

- ::= -> left part of the rule is defined to be right part of the rule
- | -> alternatively

- `::+` -> left part of the rule is completed with right part of the rule as a new alternative
- `<text>` -> nonterminal
- `"text"` -> literal
- `*` -> the preceding syntactic unit can be repeated zero or more times
- `+` -> the preceding syntactic unit must be repeated at least once
- `[]` -> the enclosed syntactic units are grouped as a single syntactic unit
- `{ }` -> the enclosed syntactic unit is optional – may occur zero or more time

IDL is a purely descriptive language. This means that invoking methods, implementing them or creating and accessing data cannot be written in IDL, but in a programming language, for which mappings from IDL constructs have been defined. A source file containing specifications written in IDL must have ".idl" extension.

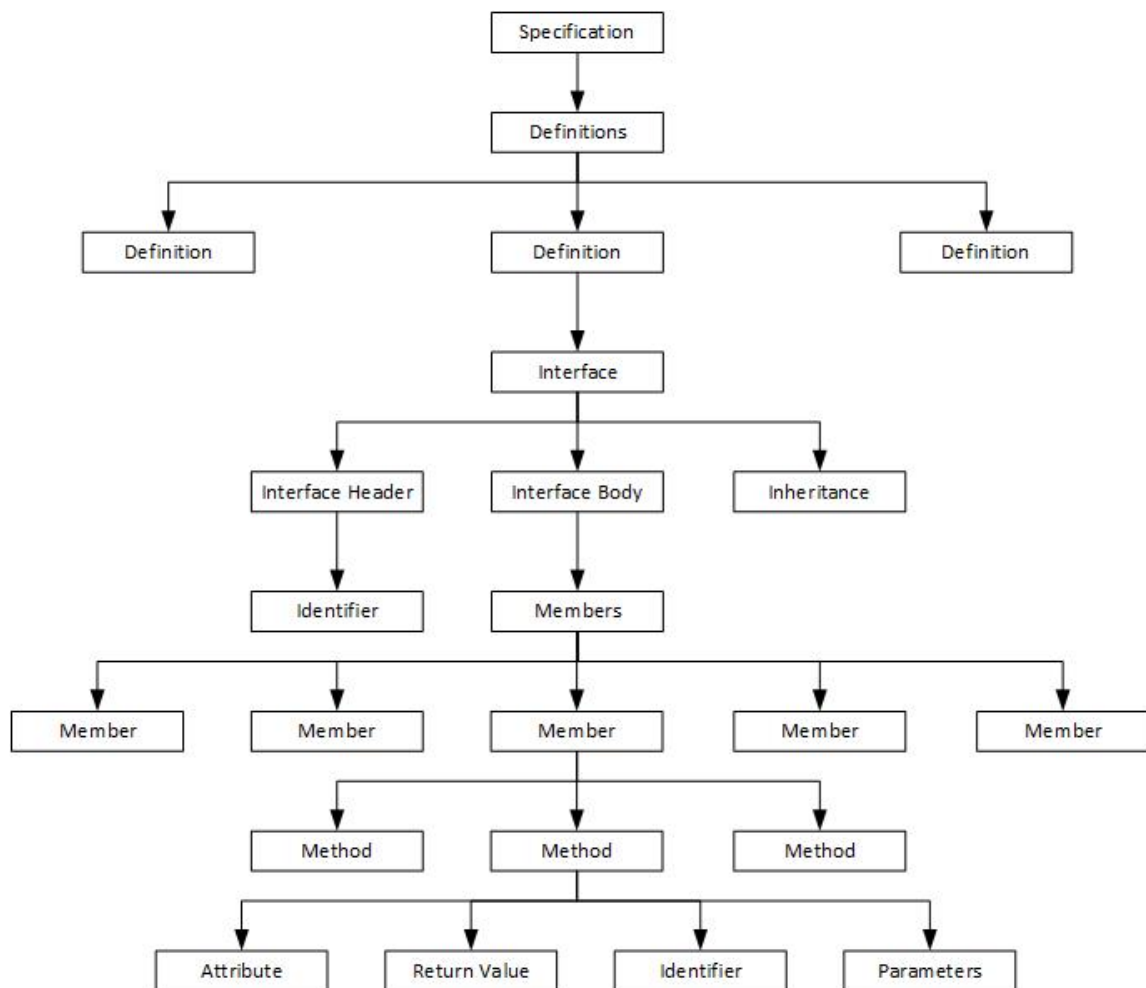


Figure 4.5: Sample abstract syntax tree of IDL grammar

Figure 4.5 shows sample AST of interface definition using IDL grammar. The root of the grammar is called specification. The specification contains definitions. Each definitions has multiple variations and interface is one of them. An interface contains interface header, interface body, and inheritance specifications. The interface header has the identifier of the interface. The

interface body has members which could be either property or method. A method contains attribute, return value, identifier, and parameters. The detailed implementation of the grammar is explained in Chapter 5.1.4.

4.2.2 Interface metamodel

The interface metamodel is intended to provide a type system for creating an interface model that are used for the code generation. A common metamodel for code generation was proposed by Michael Piefel in [21]. While the metamodel presented there is far from finished, it provides general idea to create a metamodel for our framework because they have chosen to use MOF as the meta-metamodel as well. Figure 4.6 shows generic class diagram of our interface matemodel. The metamodel is designed considering the COM technology.

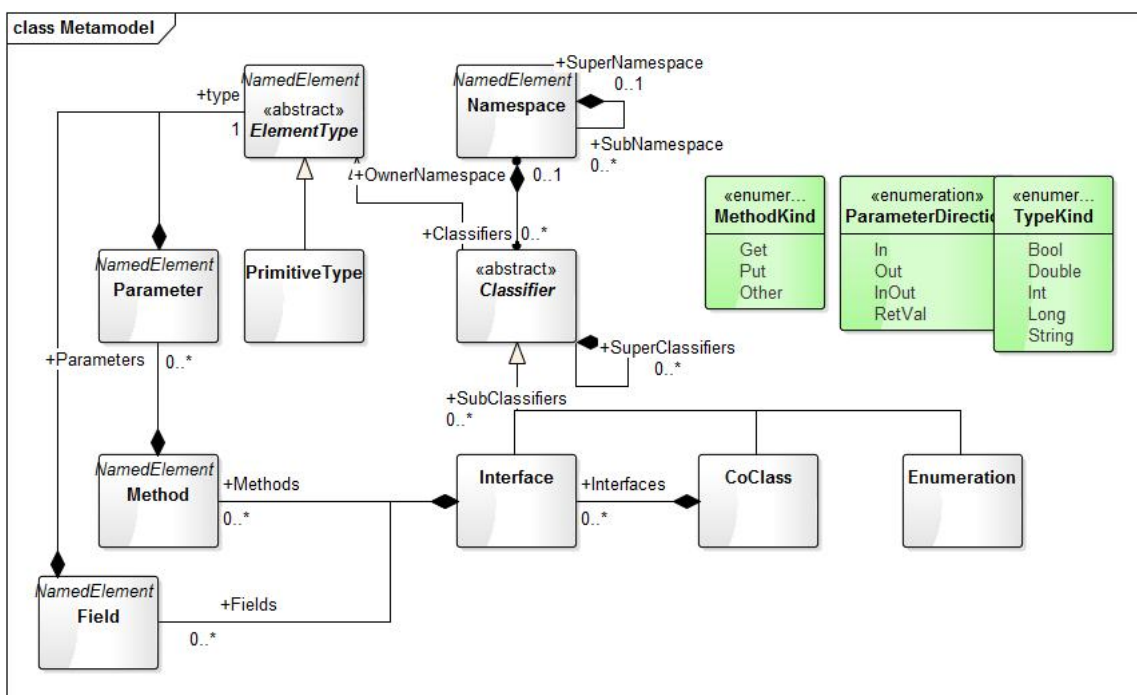


Figure 4.6: Interface metamodel design

At the root of the inheritance tree, there is the abstract class *NamedElement*, and each element in any model will be an instance of a subclass of it. The *NamedElement* adds the ability for a model element to hold a name.

There are three kind of enumerations in the metamodel: *MethodKind*, *ParameterDirection*, and *TypeKind*. The *MethodKind* enumeration defines whether a method is getter, putter, or any other type. The *ParameterDirection* enumeration defines whether a parameter is in, out, inout, or retval type. The *TypeKind* enumeration defines whether a type is bool, double, int, long, or string.

The *Namespace* is an important structural concept of the metamodel. It forms a space where elements like interfaces can reside in without colliding with elements of the same name in other namespaces. The *Namespace* is a specialization of the *NamedElement*. It has a list of classifiers and sub-namespaces, as well as a property for defining super-namespaces.

The *Classifier* is another important element of the metamodel. It has two lists for containing

super-classifiers and sub-classifiers. Three elements are specialized from the *Classifier* abstract class such as *Interface*, *CoClass*, and *Enumeration*.

In COM, a class definition consists of interface specifications. This quality of the coclass (COM Class) is reflected in the metamodel with a list of interfaces. Each interfaces has a list of methods and fields. Each methods has a list of parameters that has a type either primitive or classifier. If a method has classifier type parameters, each classifier type parameter informations are mapped to a new field instance and added to the fields list.

There are two main group of types specialized from the *ElementType*: primitive types and classifier types. Primitive types are the predefined type of object oriented languages. As a start, bool, double, int, long, and string are used as included in the *TypeKind* enumeration. Another variety of type is classifier type. Each specialized classifier elements such as interface, coclass, or enumeration are also used as a type.

4.2.3 Parser

The parser component parses input IDL models to an AST. As mentioned in the previous chapter, the Irony .Net Language Implementation Kit, which is available as an open source library, is used for this component.

Figure 4.7 shows the class diagram of the parser design. The Irony library has an abstract class *Grammar* for generic grammar definition. We have inherited the *IdlGrammar* from the abstract class and implemented the IDL grammar. The implementation details of the grammar is discussed in the Chapter 5.1.4. Furthermore, we have implemented a static extension class *IronyParserExtension*, which includes additional functionalities for parsing the input IDL models. The implementation details of the static extension class is discussed in Chapter 5.1.1.

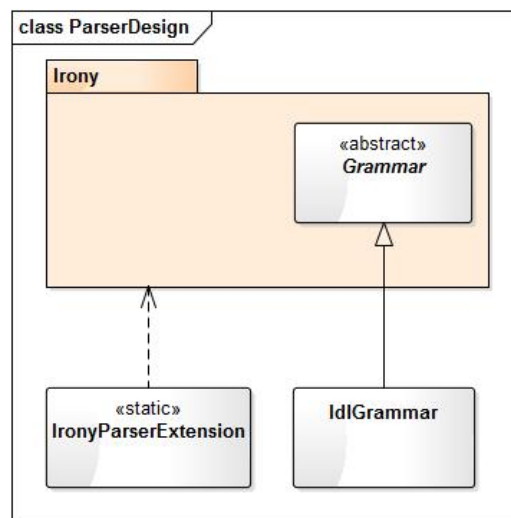


Figure 4.7: Parser design

4.2.4 Model-to-Model Transformer

The model-to-model transformer component is the corner stone of the framework. It connects the gap between two different technology spaces, such as EBNF and MOF, by mapping the relevant informations using a transformation rule. This section introduces the structural design of the model-to-model transformer and its transformation rule.

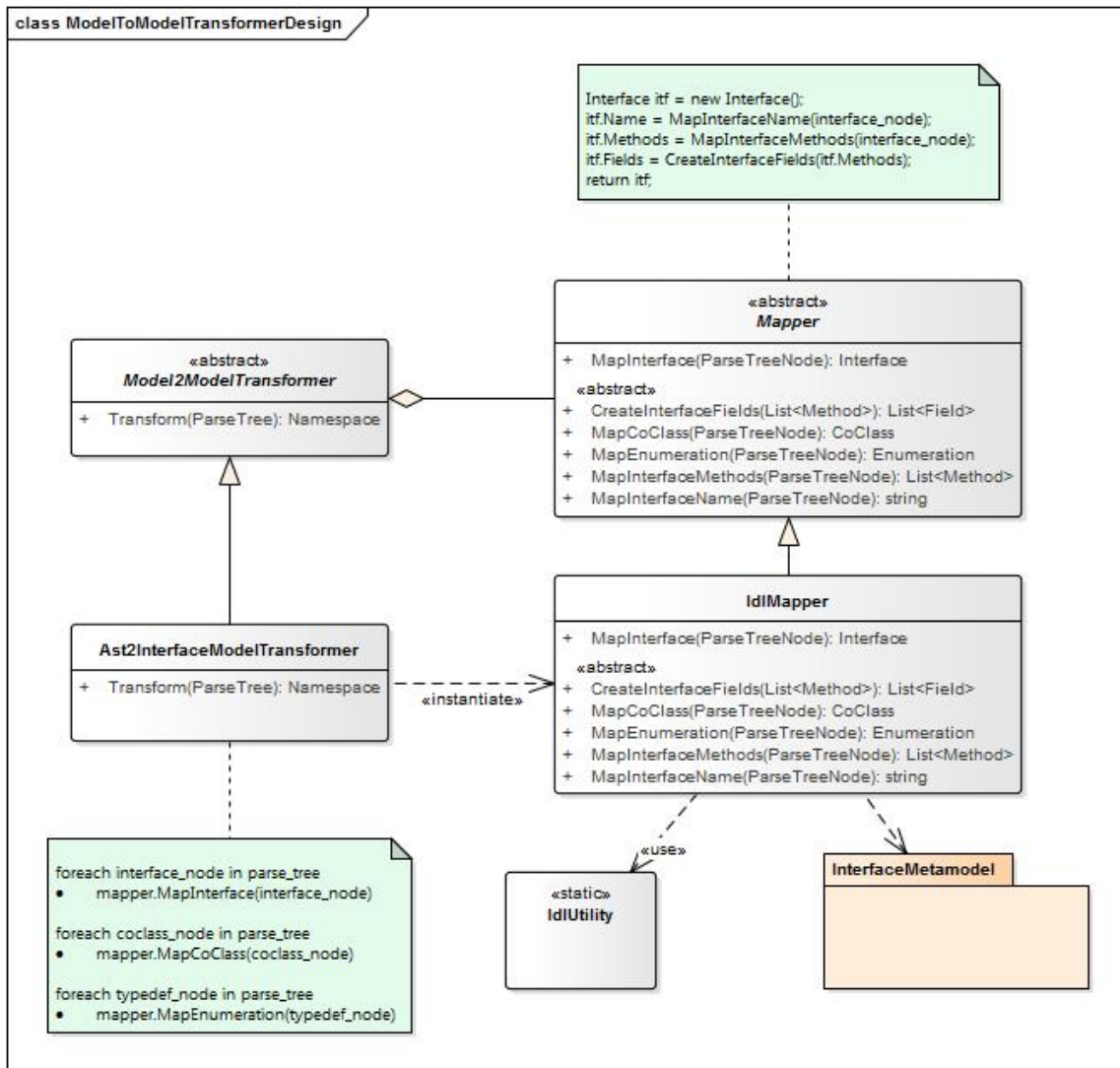


Figure 4.8: Model-to-Model transformer design

Figure 4.8 shows detailed design of the model-to-model transformer. The builder (or mapper) design pattern is used for separating the transformation of a complex interface model object from its representation so that the same transformation process can create different representations. The classes and objects participating in this pattern are:

- *Mapper*: specifies an abstract interface for mapping parts of an interface model
- *IdlMapper*: maps parts of the interface model by implementing the *Mapper* interface, defines and keeps track of the representation it creates, provides an interface for retrieving the interface model
- *Ast2InterfaceModelTransformer*: constructs an object using the *Mapper* interface
- *InterfaceMetamodel*: represents the complex interface model object under transformation
- *IdlUtility*: helps the *Mapper* class to get AST nodes according to the IDL grammar

The generic structure of the *InterfaceMetamodel* type interface contains name, methods, and fields. Hence, template method design pattern is used in the *MapInterface* method. This pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template

method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. The classes and objects participating in this pattern are:

- *Mapper*: defines abstract primitive method that concrete subclasses define to implement steps of an algorithm, implements a template method defining the skeleton of an algorithm. The template method calls primitive methods as well as methods defined in *Mapper* or those of other objects.
- *IdlMapper*: implements the primitive methods to carry out subclass-specific steps of the algorithm

Transformation rule

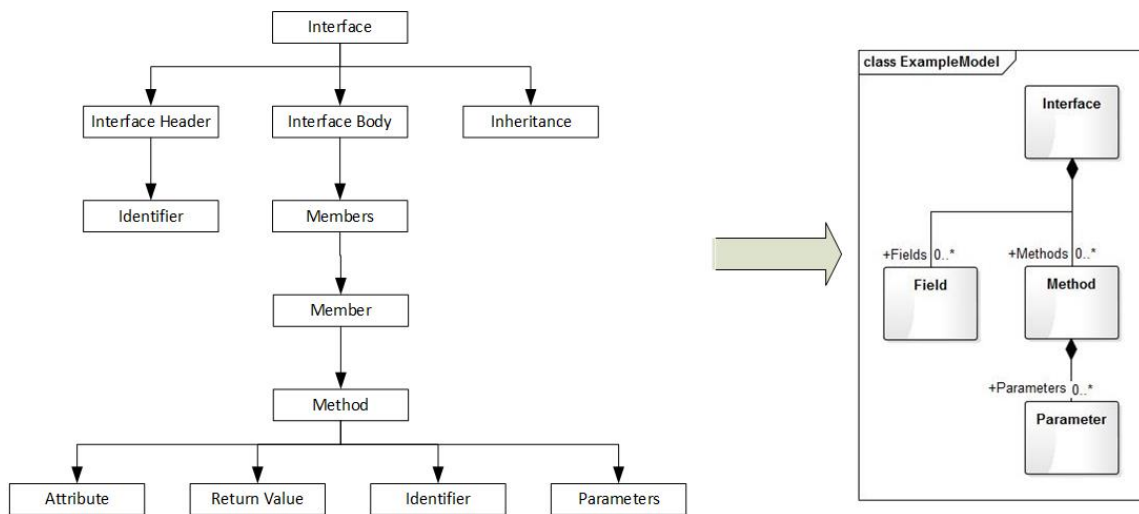


Figure 4.9: Sample AST to interface model transformation

In order to transform the AST of an IDL model to interface model (like shown in Figure 4.9), a set of transformation rule is needed. This sub-section explains the transformation rules for interface, method, and parameter transformation. The two model types are compared in tables to show the rules in an understandable fashion.

Abstract syntax tree	Interface model
Interface	Interface
+ Interface header	
—> Attributes	
—> Keyword	
—> Identifier	+ Name
+ Interface body	
—> Members	+ Methods
—> Member	
—> Method	—> Method
+ Inheritance	+ Super Classifiers

Table 4.1: Interface transformation rule

Table 4.1 shows interface transformation rule. An AST of interface contains three main nodes: interface header, interface body, and inheritance. This information is mapped to an interface

model by creating new object of interface type and assign values to its properties.

The IDL interface header node has three sub-nodes: attributes, keyword, and identifier. First sub-node IDL attributes are keywords that specify the characteristics of an interface and of the data and methods within that interface [22]. The interface header attributes represent information about the entire interface for directing the MIDL compiler what to generate from IDL file. We did not use these attributes in the scope of this project because our output interface models do not have properties or members that are directly related to the interface header attributes. The second sub-node keyword represents the type of the node. Since it is an interface node, the keyword would be *interface* in this case. We identify the node by its keyword and create new instance of interface model. The third sub-node identifier represents the name of the interface. This information is directly mapped to the *Name* property of the interface model.

The IDL interface body has a list of members where each members are method specifications. The members list matches with the methods list of the interface model. Therefore, each member/method information is mapped to the interface model's methods. The mapping of methods is presented in Table 4.2.

The inheritance specifies base class of the interface. It is mapped to the *SuperClassifier* property of the interface model.

Abstract syntax tree	Interface model
Method	Method
+ Attributes	
—> Attribute	
——> Keyword	+ Kind
+ Return value	
—> Keyword	
+ Identifier	+ Name
+ Parameters	+ Parameters

Table 4.2: Method transformation rule

Table 4.2 shows method transformation rule. An AST of method contains four main nodes: attributes, return value, identifier, and parameters. Same as before, this information is mapped to a method of the interface model by creating new instance of method and assign values to its properties.

The IDL method attribute contains keyword for defining the method kind such as *propget* and *propput*. This information is mapped to method kind property of the interface model. If the attribute value is *propget* the interface model type method becomes "get" and if the attribute value is *propput* the interface model type method becomes "put".

The return value of all IDL methods are *HRESULT*. Hence, this information is not used in interface model. The identifier of the IDL method is mapped to the *Name* property of the interface model method.

In IDL each method has a list of parameters. These parameters are mapped directly to the parameters list property of the interface model type method. The mapping details are presented in Table 4.3.

Abstract syntax tree	Interface model
Parameter	Parameter
+ Attributes	
—> Attribute	
——> Keyword	+ Direction
+ Type	
—> Keyword	+ Element type
+ Pointer	
—> Key symbol	+ Pointer
+ Identifier	+ Name

Table 4.3: Parameter transformation rule

Table 4.3 shows parameter transformation rule. An IDL parameter contains four main nodes: attributes, type, pointer, and identifier. The information is mapped to an interface model parameter by creating new instance of parameter and assigning values to its properties.

The IDL parameter attribute contains an information about the direction of the method such as in, out, inout, or retval. This information is directly mapped to the *Direction* property of the interface model type parameter because the interface model has also the same direction types.

The IDL parameter has two types: primitive and classifier. The primitive types are generic types like bool, double, float, int, and string. The classifier type is one of the other interfaces declared in the same IDL file or imported IDL file. The information is mapped to the type property of the interface model type parameter. The type property is a type of *Element Type* which is also specialized in two classes for distinguishing primitive and classifier types.

The IDL pointer has symbols for representing the parameter pointers. They are directly mapped to the pointer property of the interface model parameter.

The IDL identifier contains the parameter name. Therefore, it is directly mapped to the *Name* property of the interface model parameter.

4.2.5 Model-to-Code Transformer

The model-to-code transformer component uses an interface model information to initialize wrapper code templates and write all text content of the templates to a file from the templates. As mentioned in the previous chapter, the Microsoft T4 is used as the template based code generation technology for this component.

Figure 4.10 shows detailed design of the model-to-code transformer. The *Model2CodeTransformer* is an abstract class that we use in the factory method as an interface for creating concrete implementation instance. It has protected fields for defining COM header, COM source, C++ interface, template type, as well as a dictionary for storing all templates. The concrete implementation is done in *Com2CppWrapperCodeGenerator* class, which initializes the *templateType* field in its constructor and overrides the *InitializeTemplates* method of the abstract class. The *InitializeTemplates* method initializes the wrapper code templates depending on a template type that is configured in the command line option of the custom build tool. The *Generate* method writes all text contents of the templates to files. The code generation activity is considered as generic for all templates. Thus, the *Generate* method is implemented in the abstract class. The implementation details of two methods are discussed in Chapter 5.1.3.

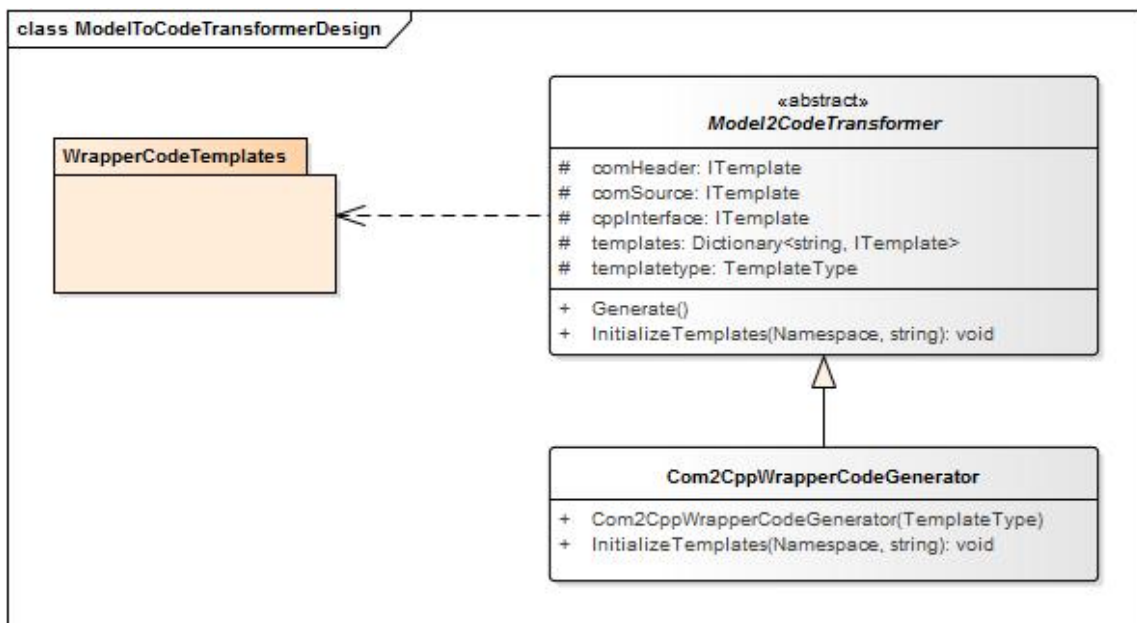


Figure 4.10: Model-to-Code transformer design

Chapter 5

Implementation

This chapter elaborates on the implementation of the framework design. The following sections discuss the implementation of parser, model-to-model transformer, model-to-code transformer, interface grammar, interface model, wrapper code templates, as well as custom build tool that we have configured in Visual Studio tool.

5.1 Implementation of the framework design

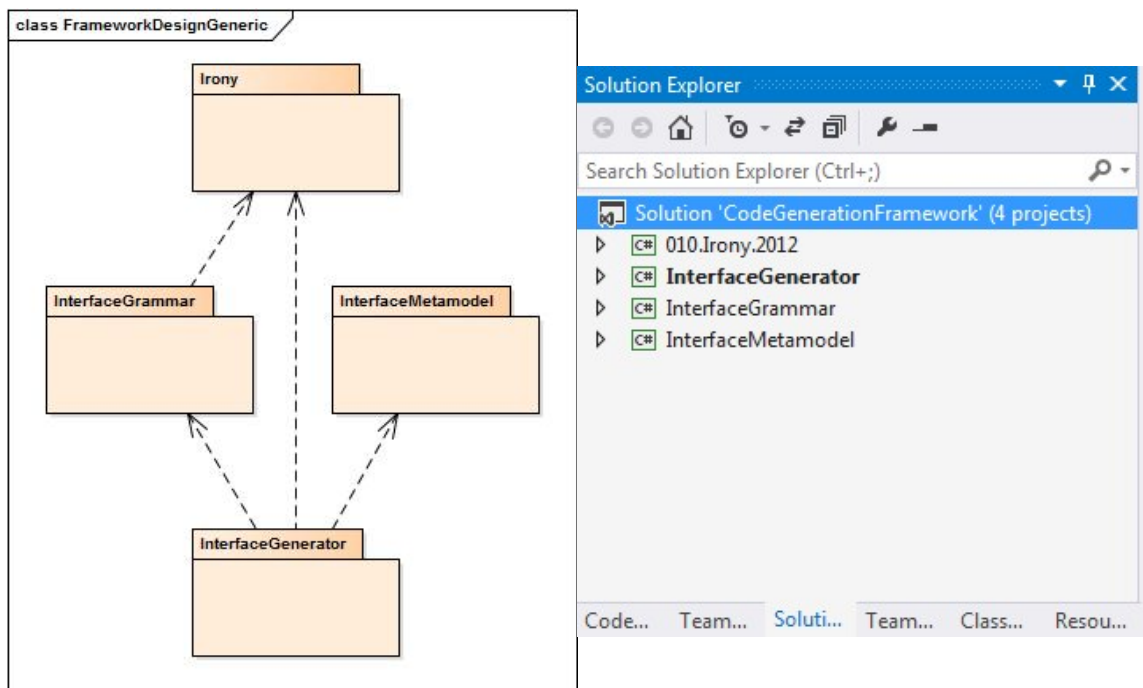


Figure 5.1: Implementation of the framework design

The implementation of the framework design is shown in Figure 5.1 as *CodeGenerationFramework* C# command line application solution in Visual Studio. The solution includes Irony library, interface grammar and interface metamodel namespaces as well as interface generator namespace.

The Irony .NET Language Implementation Kit is included in the solution as an open source library. The *InterfaceGrammar* namespace contains all DSL grammars of the framework. The

grammars are specialized from the *Grammar* class of the Irony library. The *InterfaceMetamodel* namespace contains all metamodels of the framework. The *InterfaceGenerator* namespace contains all classes of the framework design as well as wrapper code templates.

5.1.1 Parser

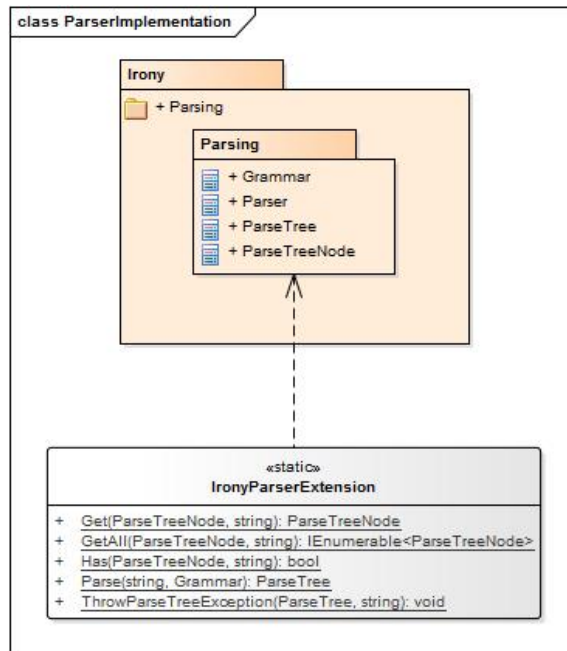


Figure 5.2: Class diagram of the parser implementation

The Irony open source library is extended using a static class which is *IronyParserExtension*. The static class make use of four particular classes from the *Irony.Parsing* namespace: *Grammar*, *Parser*, *ParseTree*, and *ParseTreeNode*. As shown in Figure 5.2, the static class has five static methods:

- *Get*: searches a node from a parse tree using the key string parameter and returns the matching node
- *GetAll*: searches for all nodes matching the key string parameter and returns all of them as a list
- *Has*: searches a node from a parse tree using the key string parameter and returns boolean value which indicates whether the tree has the node
- *Parse*: reads all text from a file, parses the text to a parse tree, and returns the tree
- *ThrowParseTreeException*: throws exception when the tree has errors or does not have a root when parse

5.1.2 Model-to-model transformer

The class diagram of the model-to-model transformer design is presented in Figure 4.8 in Chapter 4.2.4. As presented in that diagram, the *Model2ModelTransformer* class has *Transform* method which is implemented in *Ast2InterfaceModelTransformer*. Activity diagram of the *Transform* method is shown in Figure 5.3.

There are two types of IDL files are identified during the project. First type of IDL files include a library definition that has definitions of members such as interfaces, coclasses, and

enumerations. Second type of IDL files include directly defined members without collecting them in a library. Thus, the *Transform* method has two types of mapping procedures for the two IDL file content types respectively.

The first procedure is for the IDL files that contain a library definition. If the IDL file has a library definition, the *Transform* method gets the library node from its parse tree and maps the library node information to a namespace. Then, it maps each member nodes of the library to the matching interface models and adds the interface models to the namespace.

The second procedure is for the IDL files that contain member definitions directly. The *Transform* method maps each member nodes of a parse tree to the matching interface models and adds the interface models to a namespace. The namespace is initialized using the same name as the member node.

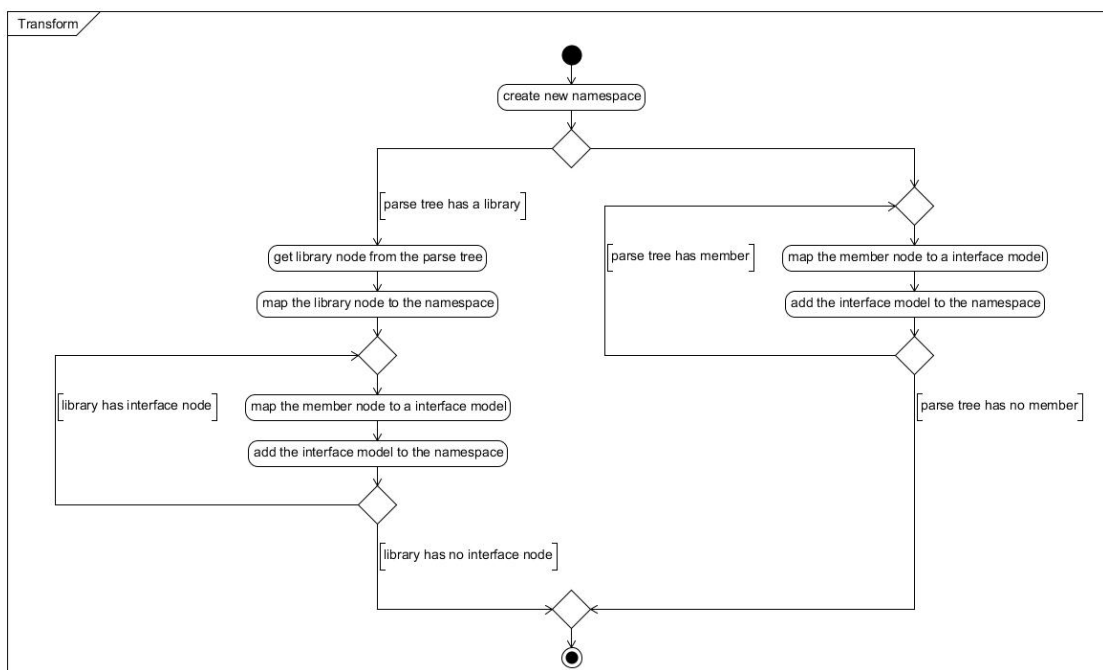


Figure 5.3: Activity diagram of Transform method

The *Transform* method uses the *Mapper* class for mapping the member nodes of a parse tree to an object-oriented interface model type instances. The *IdlMapper* class implements abstract methods of the *Mapper* abstract class such as:

- *CreateInterfaceFields*: Checks all parameters of a given method whether they are primitive or classifier type. If the method has classifier type parameters, creates new fields using the parameters information and returns a list of fields.
- *MapCoClass*: Maps given coclass node information to an interface model type coclass object and returns the coclass object.
- *MapEnumeration*: Maps given typedef node information to an interface model type enumeration object and returns the enumeration object.
- *MapInterfaceMethods*: If given interface node has methods in its body, maps all methods to an interface model type method objects and returns a list of methods.

- *MapInterfaceName*: Returns the name of given interface node.

Additionally, the *IdlMapper* class has private methods for internal use such as:

- *MapMethod*: Maps given method node information to an interface model type method object and returns the method object. The mapping procedure includes name, attributes, and parameters mapping activities. The name mapping activity is a direct assignation from the method node identifier information to the interface model type method object. The attributes mapping activity involves assignation of the method kind such as get, put, or other. The parameters mapping activity uses the following private method *MapParameter*.
- *MapParameter*: Maps given parameter node information to an interface model type parameter object and returns the parameter object. The mapping procedure include type, name, pointer, and attributes mapping activities. The type mapping activity varies to primitive and classifier type mappings depending on whether the parameter node is primitive or classifier type. The following two methods, *MapClassifierTypeParameter* and *MapPrimitiveTypeParameter*, are used for each. The name mapping activity is a direct assignation from the parameter node identifier information to the interface model type parameter object. The pointer mapping activity is assignation of boolean fields of the interface model type parameter. If the parameter node has the symbol "*" or "***" the boolean fields are set to true accordingly. The attribute mapping activity involves assignation of the parameter direction such as in, out, inout, or retval. If the parameter has both out and retval attributes, the direction would set to *outretval*.
- *MapClassifierTypeParameter*: Creates new classifier type parameter object using given name and returns the parameter object.
- *MapPrimitiveTypeParameter*: Creates new primitive type parameter object using given name and return the parameter object. The parameter type kind varies depending on the given name such as bool, char, int, float, double, void, or string.

5.1.3 Model-to-code transformer

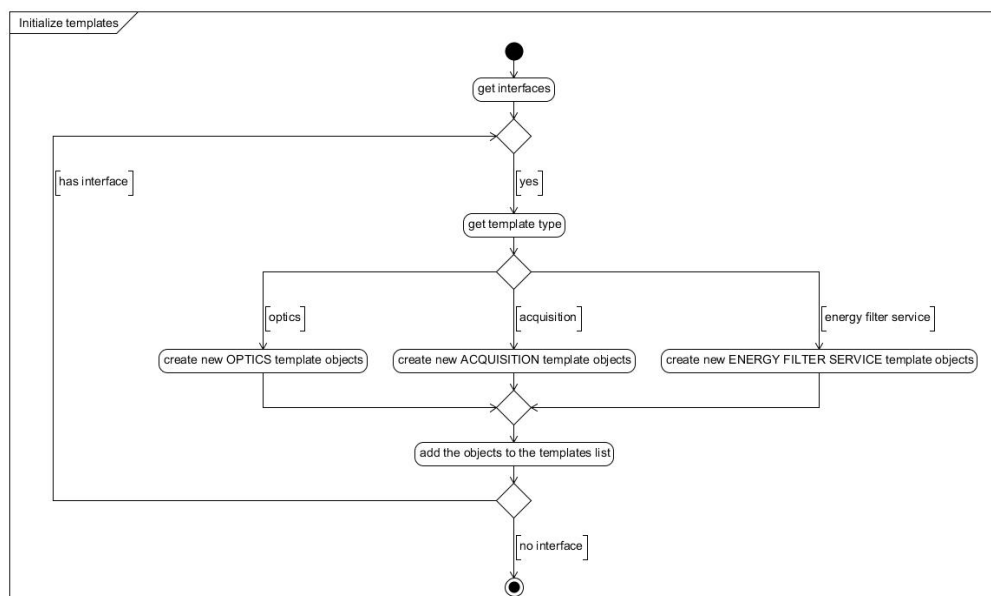


Figure 5.4: Activity diagram of the InitializeTemplates method

The class diagram of the model-to-code transformer is presented in Figure 4.10 in Chapter 4.2.5. As presented there, the class has two methods: *InitializeTemplates* and *Generate*. This section introduces each methods with their activity diagrams.

The *Com2CppWrapperCodeGenerator* concrete class implements the *InitializeTemplates* abstract method of the *Model2CodeTransformer* abstract class. Figure 5.4 shows activity diagram of the *InitializeTemplates* method. A template type is given from the command line option of the custom build tool. The command line option currently has three options: optics, acquisition, and energy filter service. The *InitializeTemplates* method uses the option to create suitable template objects and adds them to the template list of the base class, *Model2CodeTransformer*.

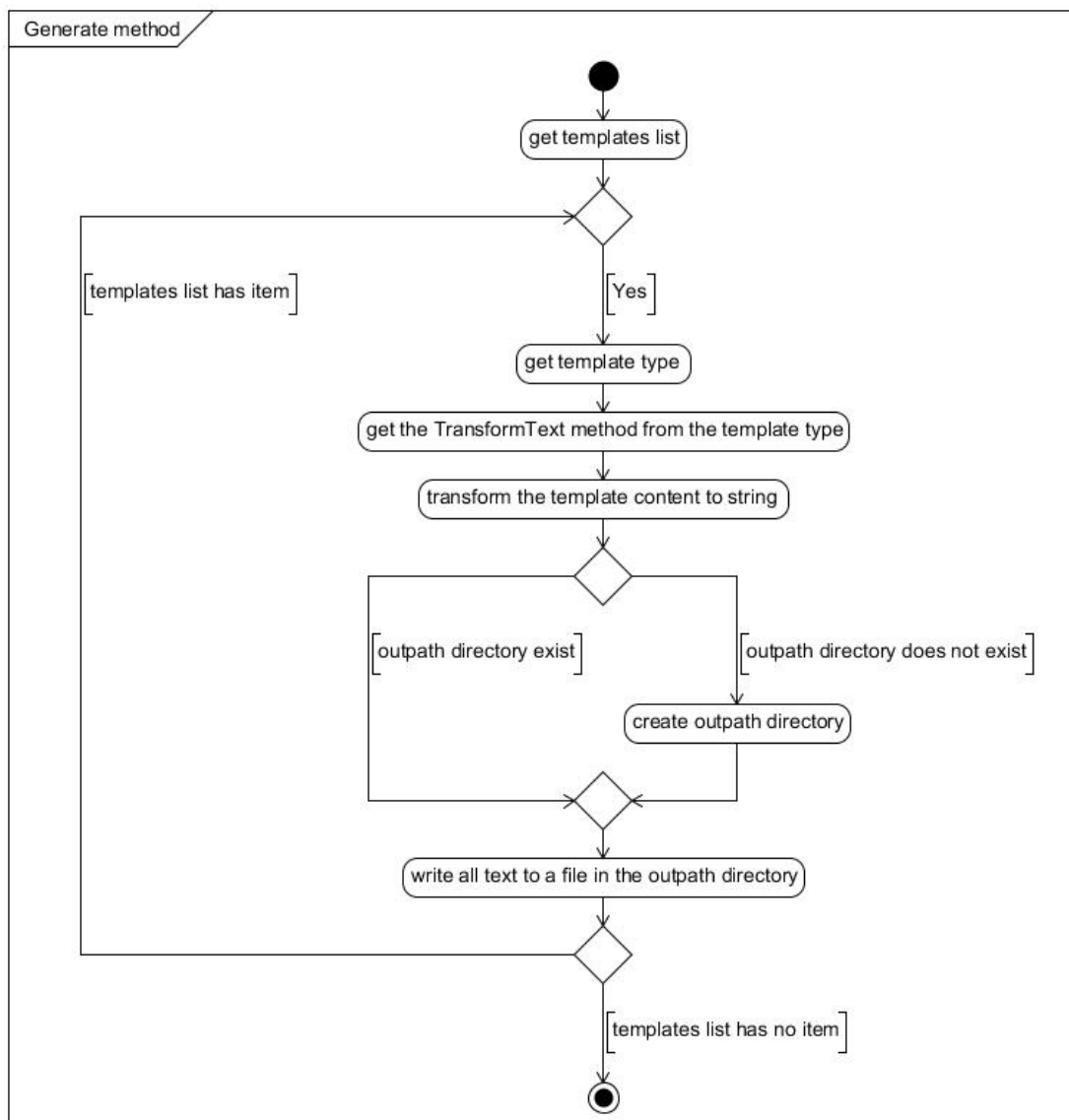


Figure 5.5: Activity diagram of the Generate method

The *Generate* method is implemented in the *Model2CodeTransformer* abstract class. Figure 5.5 shows activity diagram of the *Generate* method. The methods gets all items of the templates list, converts each template's contents to string, and writes the string text to a file in given outpath

directory.

5.1.4 Interface grammar

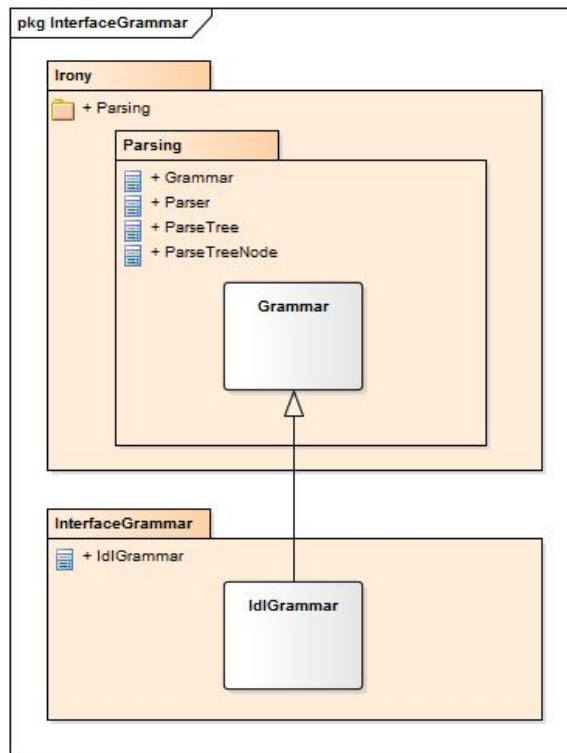


Figure 5.6: Class diagram of the interface grammar implementation

In *InterfaceGrammar* namespace, the IDL grammar is implemented in the *IdlGrammar* class which is derived from the *Grammar* class of the *Irony.Parsing* namespace, see Figure 5.6. The implementation of IDL grammar has three main sections: terminals, nonterminals, and rules.

Terminal and nonterminal symbols are lexical elements used in specifying the rules constituting the IDL grammar. Terminal symbols are the elementary symbols of the IDL. Nonterminal symbols are replaced by groups of terminal symbols according to the rules.

In our implementation, the terminals include regex based terminal for GUID, number literals for integer and float, as well as other literals. Also, the terminals include lexical structure such as string literal, char literal, number, identifier, comment, and other symbols like colon, semicolon, dot, comma, question mark, and brackets. The following code snippet shows sample terminal symbol declaration using Irony library in C# for GUID, integer, float, and other literals:

```

...
...
...
RegexBasedTerminal uuid_rep = new RegexBasedTerminal("uuid", "[A-Fa-f0-9]{8}-[A-Fa-f0-9]{4}-[A-Fa-f0-9]{4}-[A-Fa-f0-9]{4}-[A-Fa-f0-9]{12}");
NumberLiteral integer_literal = new NumberLiteral("integer") { Options = NumberOptions.IntOnly | NumberOptions.AllowSign | NumberOptions.Hex | NumberOptions.AllowLetterAfter };
NumberLiteral float_literal = TerminalFactory.CreateCSharpNumber("float");
var other_literal = new RegexBasedTerminal("[^\\t\\n\\r 0-9A-Z_a-z]");
...
...
...

```

The nonterminals include definitions of nodes that are used for defining the grammar rule. In Irony, nonterminals are implemented as *NonTerminal* type instances. The following code snippet shows an example declaration of nonterminals in our IDL grammar:

```
NonTerminal specification = new NonTerminal("specification"),
    definitions = new NonTerminal("definitions"),
    definition = new NonTerminal("definition"),
    ...
    interface = new NonTerminal("interface"),
    interface_header = new NonTerminal("interface_header"),
    interface_body = new NonTerminal("interface_body"),
    interface_body_member = new NonTerminal("interface_body_member"),
    interface_body_members = new NonTerminal("interface_body_members"),
    ...
    method = new NonTerminal("method"),
    ...
```

In Irony, grammar rule is implemented as an EBNF expression of nonterminal symbol. The implementation of the IDL grammar using the Irony is as follows:

```
specification.Rule = definitions;
definitions.Rule = MakeStarRule(definitions, definition);
definition.Rule = interface | ...;
...
interface.Rule = interface_header + inheritance_spec + interface_body + ToTerm(';');
interface_header.Rule = attribute_specification + "interface" + identifier;
inheritance_spec.Rule = ToTerm(':') + scoped_names;
...
interface_body.Rule = ToTerm('{') + interface_body_members + '}';
interface_body_members.Rule = MakeStarRule(interface_body_members, interface_body_member);
interface_body_member.Rule = method | ...;
...

```

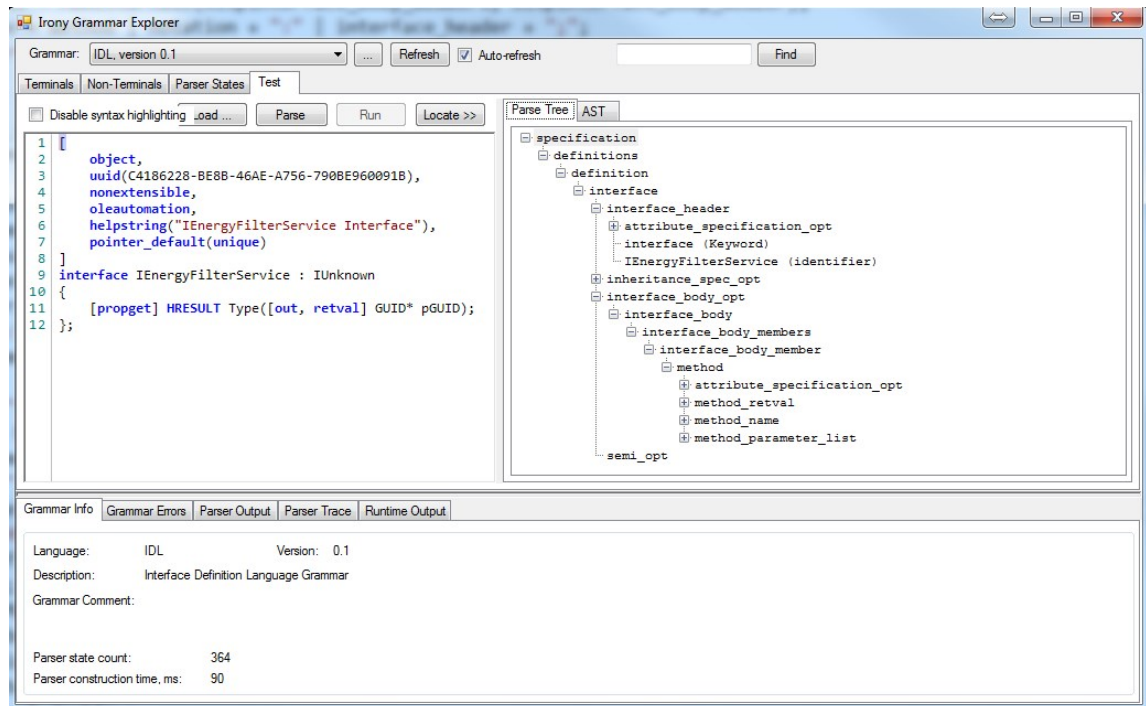


Figure 5.7: Sample interface and its parse tree

The Irony .NET Language Implementation Kit comes with a grammar explorer tool in the open source package for verifying the developed grammar. The tool is intended to parse given text file

conforming to loaded grammar into parse tree. Also, the tool identifies grammar errors which helps the grammar developer to easily fix if there are any conflicting rules in the grammar. Figure 5.7 shows sample IDL interface model and its parse tree on the Irony Grammar Explorer tool using our IDL grammar implementation.

5.1.5 Interface metamodel

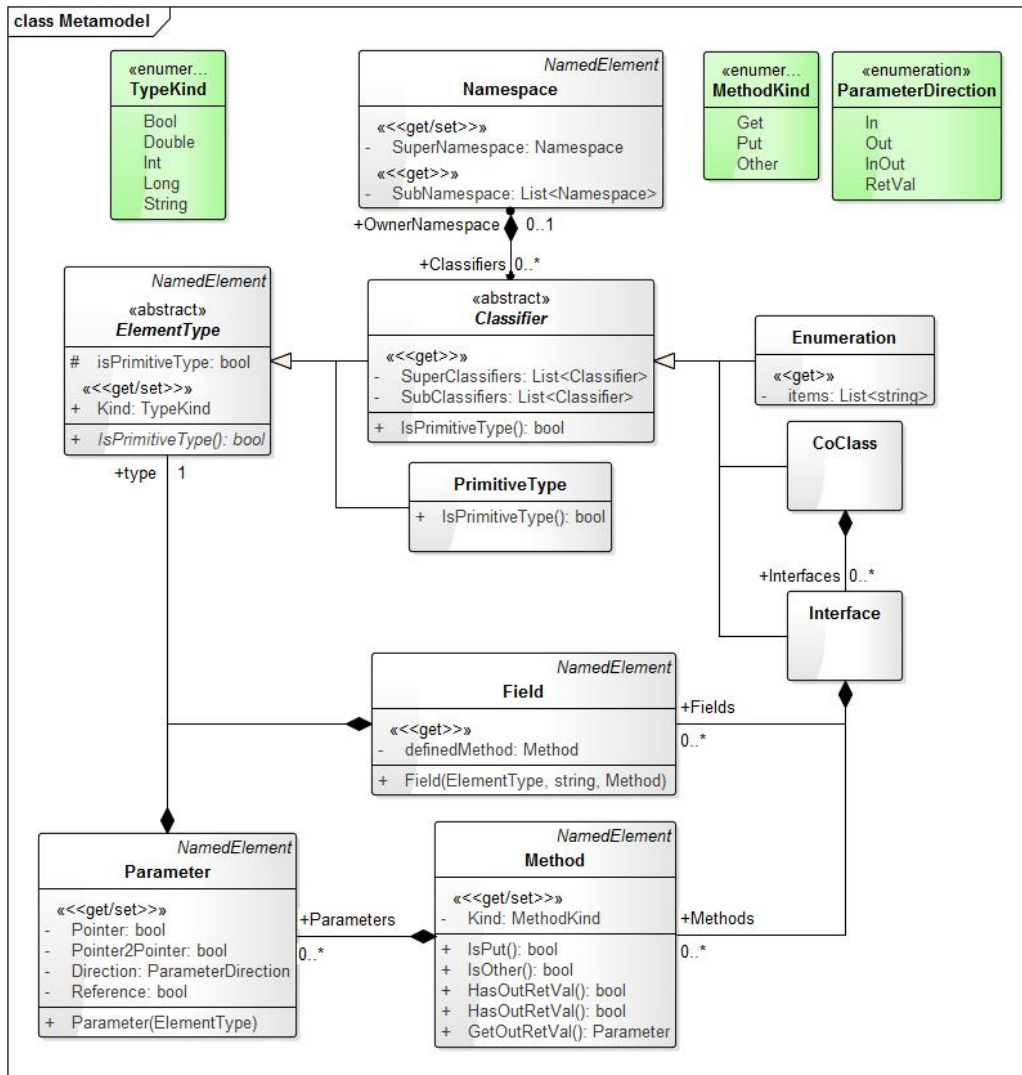


Figure 5.8: Detailed class diagram of interface metamodel

Figure 5.8 shows detailed class diagram of interface metamodel that we have introduced in Chapter 4.2.2. This class diagram is the direct representation of the actual C# implementation including all private, protected, and public fields and methods. Hence, the diagram is considered as self explanatory.

5.1.6 Wrapper code templates

In order to separate template abstraction from its actual implementation, we provide a generic interface for all templates. This interface is an agreement between the model-to-code transformer and the wrapper code templates. Thus, every template implements the *ITemplate* interface. In order to generate the COM-to-C++ wrapper code, we needed three code artifacts such as COM

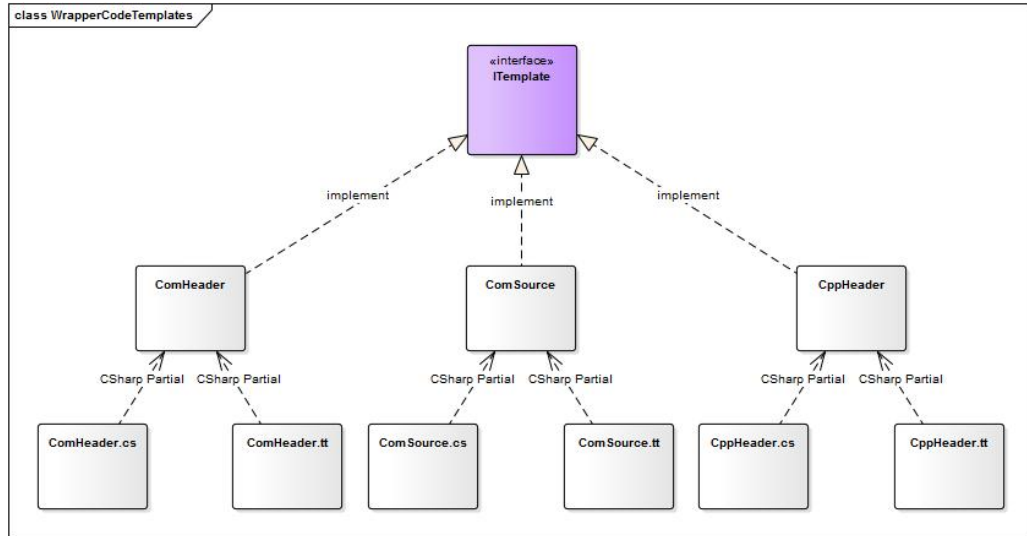


Figure 5.9: Class diagram of the wrapper code templates

interface, COM implementation, and C++ interface. Hence, we have created three templates for the artifacts: *ComHeader*, *ComSource*, and *CppHeader*, see Figure 5.9. Each template consists of two type of partial classes: C# class and run-time T4 template class. The C# class has helper methods for the template. The run-time T4 template has the body of the code.

```

<#@ include file="common.tt" #>
// Copyright (c) 2009 - 2013 by FEI Company
// All rights reserved. This file includes confidential and proprietary information of FEI Company.

#pragma once

namespace Fei {
namespace Tem {
namespace Iom {

<#> foreach (var fieldType in itf.Fields.Select(x => x.GetFieldType.Name).Distinct()) { #>
class <#= fieldType #>;
<#> #>

class I<#= itf.Name #>
{
public:
<#> foreach (var method in itf.Methods)
{
if(method.HasOutRetVal()) {
var outRetValType = method.GetOutRetVal().GetParameterType;
if (outRetValType.IsPrimitiveType()) {
#> <#= GetPrimitiveTypeIdentifier(outRetValType) #><#>
}
else {
#> <#= outRetValType.Name #><#>
} #><#>
} #> <#= method.Name #>() const = 0;
<#> #>
};

} // namespace Iom
} // namespace Tem
} // namespace Fei
  
```

Directives

Text blocks

Control blocks

Figure 5.10: Sample T4 template for generating C++ interface

Figure 5.10 shows sample T4 template. The template contains the text that will be generated from it. There are various control blocks inserted in the template. The control blocks are fragments of program code. They provide varying values and allow parts of the text to be conditional and repeatable. This structure makes a template easy to develop because the developer can start with a prototype of the generated file, and incrementally insert control blocks that vary the result. The text templates are composed of the following parts [23]:

- **Directives** - elements that control how the template is processed
- **Text blocks** - content that is copied directly to the output
- **Control blocks** - program code that inserts variable values into the text, and controls conditional or repeated parts of the text

Control blocks are sections of program code that are used to transform the templates. A standard control block is a section of program code that generates part of the output file. Any number of text blocks and standard control blocks can be mixed in a template file. However, a control block cannot be placed inside another. Each standard control block is delimited by the symbols `<# ... #>`. An expression control block evaluates an expression and converts it to a string. This is inserted into the output file. Expression control blocks are delimited by the symbols `<#= ... #>`.

5.2 Custom build tool

A custom build tool was configured during the project for generating COM-to-C++ wrapper code artifacts from an IDL file. The tool has functionalities of both the standard MIDL compiler and our code generation framework. The MIDL compiler generates a TLB files from the input IDL file. The code generation framework generates COM-to-C++ wrapper code from the input IDL file.

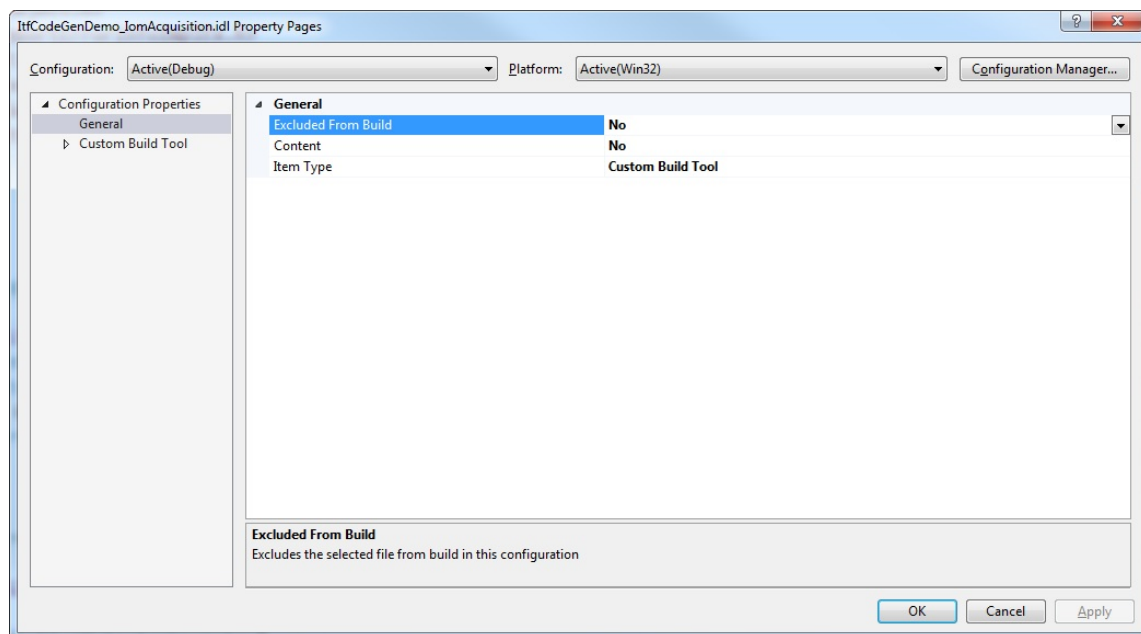


Figure 5.11: Visual Studio custom build tool option

Figure 5.11 shows Visual Studio property page for configuring a custom build tool for a file. A custom build tool provides the build system with the information it needs to build specific input files [24]. It specifies a command to run, a list of input files, a list of output files that are generated by the command, and an optional description of the tool.

The custom build tool is specified in the **Property Pages** dialog box of the ".idl" file. The item type is selected as *Custom Build Tool* instead of *MIDL*. The selected file must be included in build. Hence, the "Exclude From Build" option is set as "No".

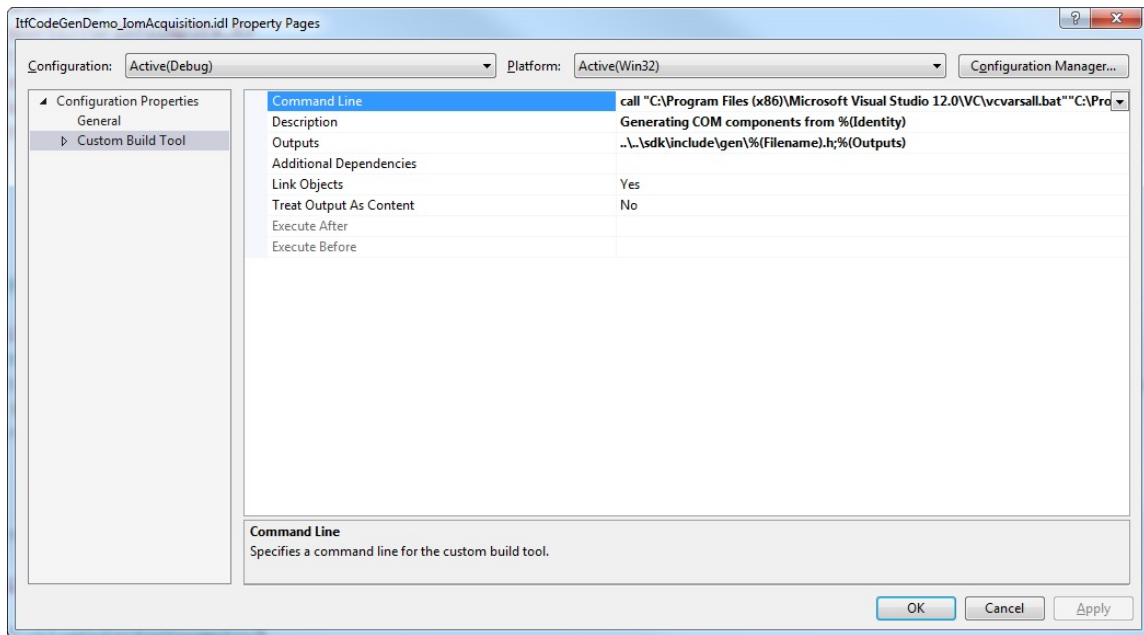


Figure 5.12: Visual Studio custom build tool option

Figure 5.12 shows detailed configuration of the custom build tool. In **Command Line**, the following command is specified:

```
call "C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\vcvarsall.bat"
"C:\Program Files (x86)\Windows Kits\8.1\bin\x86\midl.exe" /D _DEBUG /D _UNICODE /I..\gen\
/I..\shared\include\gen\ /I..\..\shared\include\gen\ /I..\..\sdk\include\gen\ /I..\
shared\include\idl\ /I..\..\shared\include\idl\ /I..\..\sdk\include\idl\ /I..\..\..\
FEI_CPPLIBS\sdk\include\idl\ /I..\..\..\COMMON_TYPES\sdk\include\idl\ /I..\..\..\
IMAGING\sdk\include\idl\ /I..\..\..\IMAGING_TEM\sdk\include\idl\ /I..\..\..\INFRA\sdk
\include\idl\ /I..\..\..\FEI_CPPLIBS\sdk\include\gen\ /I..\..\..\COMMON_TYPES\sdk\
include\gen\ /I..\..\..\IMAGING\sdk\include\gen\ /I..\..\..\IMAGING_TEM\sdk\include\
gen\ /I..\..\..\INFRA\sdk\include\gen\ /W1 /nologo /char signed /env win32 /Oicf /out
"gen" /h "....\..\sdk\include\gen\%(Filename).h" /iid "....\..\sdk\include\gen\%(
Filename)_i.c" /tlb "....\..\sdk\include\gen\%(Filename).tlb" /no_robust %(Filename)
.idl
"%(FdtBinDebug)\InterfaceGenerator" -efs "%(FullPath)"
```

The commands are specified as if they were being specified at the command prompt. It includes a valid command or batch file, and any required input or output files. The **call** batch command is specified before the name of a batch file to guarantee that all subsequent commands are executed. The above command first calls the MIDL compiler to generate type libraries from the given IDL file. Second, the command calls the code generation framework with command line option to generate wrapper code.

The **Output** specifies the name of the output file. This is a required entry. Without a value for this property, the custom build tool will not run. If a custom build tool has more than one output, the names would be separated with a semicolon.

The name of the output file should be the same as it is specified in the **Command Line** property. The project build system will look for the file and check its date. If the output file is newer than the input file or if the output file is not found, the custom build tool would run. If the input file is older than the file specified in the **Output** property, the custom build tool would not run.

Chapter 6

Deployment

This chapter introduces the deployment of our framework considering company built environment, the framework component, and the demo project component.

6.1 Company built environment

The client company uses the component oriented code architecture (COCA) for organizing their code base. Figure 6.1 shows sample structure of the COCA. The structure consists of sandbox, component, function groups, module, and test. The sandbox contains loaded components as root folders. The component contains loaded source files from version control system. The function groups group modules of similar or related functionality. The module contains project file and sources. The test is an optional folder, which contains test code of the module. All packages are required to comply with the COCA structure, which enables easy integration of components into projects.

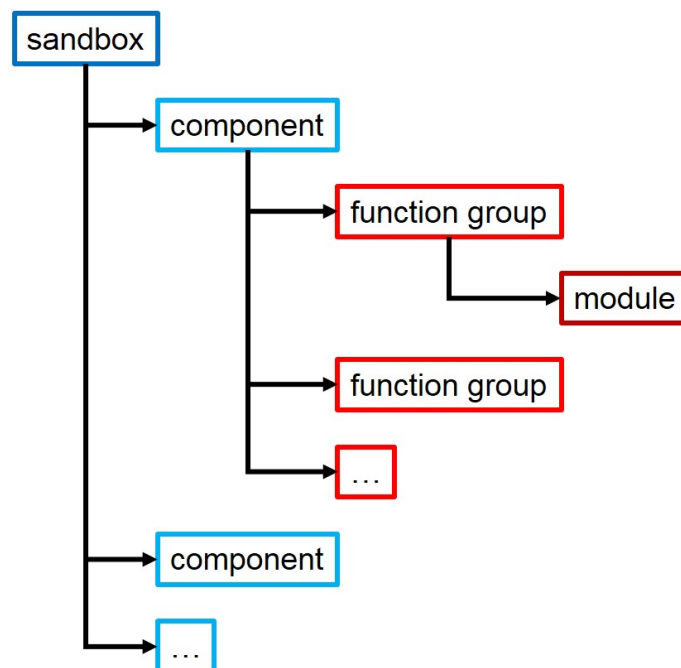


Figure 6.1: Structure of Component Oriented Code Architecture (COCA)

The build infrastructure consist of version control system, build virtual machine, holding area, and SCOTS (Shared Components Off The Shelf) server as shown in Figure 6.2. The build flow of

the of a component is as follows:

1. Load sources from version control system to build virtual machine
2. Synchronize dependencies from holding area and SCOTS server to build virtual machine
3. Build the source in build virtual machine
4. Upload results to holding area

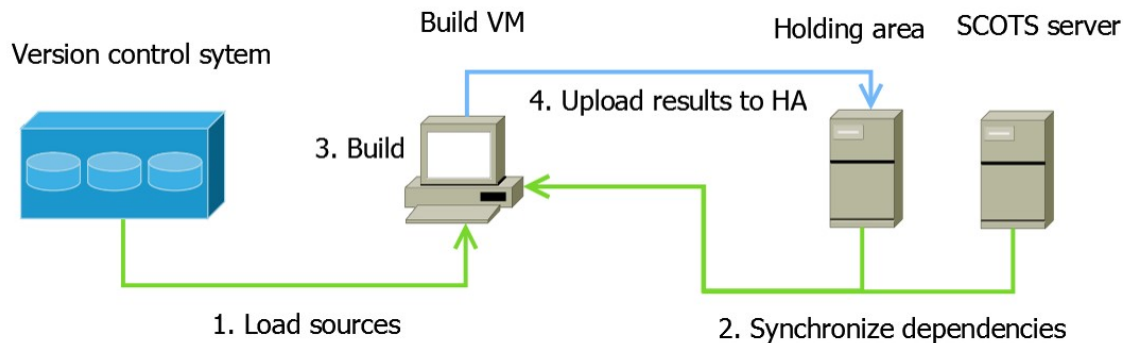


Figure 6.2: Build flow

Holding area contains copy of all successful build results. Every component build results are stored under `packages/packagesFDT` folder.

6.2 CodeGenerationFramework component

A new component, *CodeGenerationFramework*, has been created and deployed on the company build environment as the prototype of the project. Figure 6.3 shows COCA structure of the *CodeGenerationFramework* component. The component consists of the *Irony* open source library, as well as *InterfaceGenerator*, *InterfaceGrammar*, and *InterfaceMetamodel* namespaces. The shared folder is used for the build infrastructure.

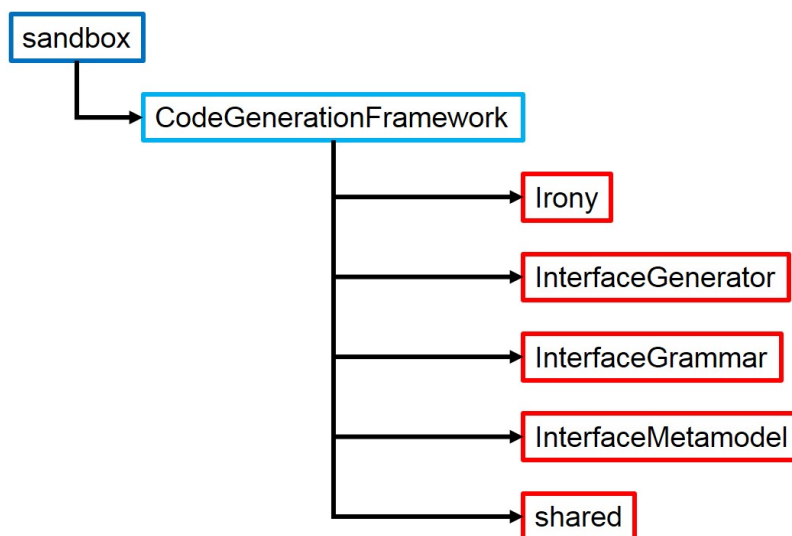


Figure 6.3: COCA structure of CodeGenerationFramework

The successful build results of the component are stored in the holding area (see Figure 6.4) which is accessible to all developers who want to consume the *CodeGenerationFramework* in their development. The results are labeled automatically using the `PRODUCT_VERSION_POSTFIX.RANDOM` form. The `POSTFIX` varies `DEV`, `RC`, and `REL` depending on whether the component is development or release version.

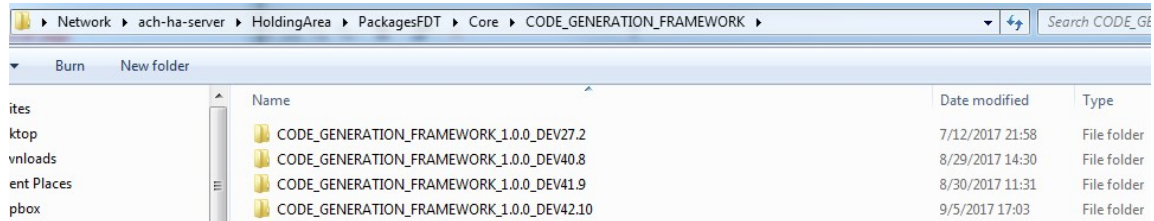


Figure 6.4: Jenkins build result stored in the Holding Area

The Jenkins open source automation server is used for building, deploying, and automating the project. Figure 6.5 shows build process of the *CodeGenerationFramework* component in the company build virtual machine.

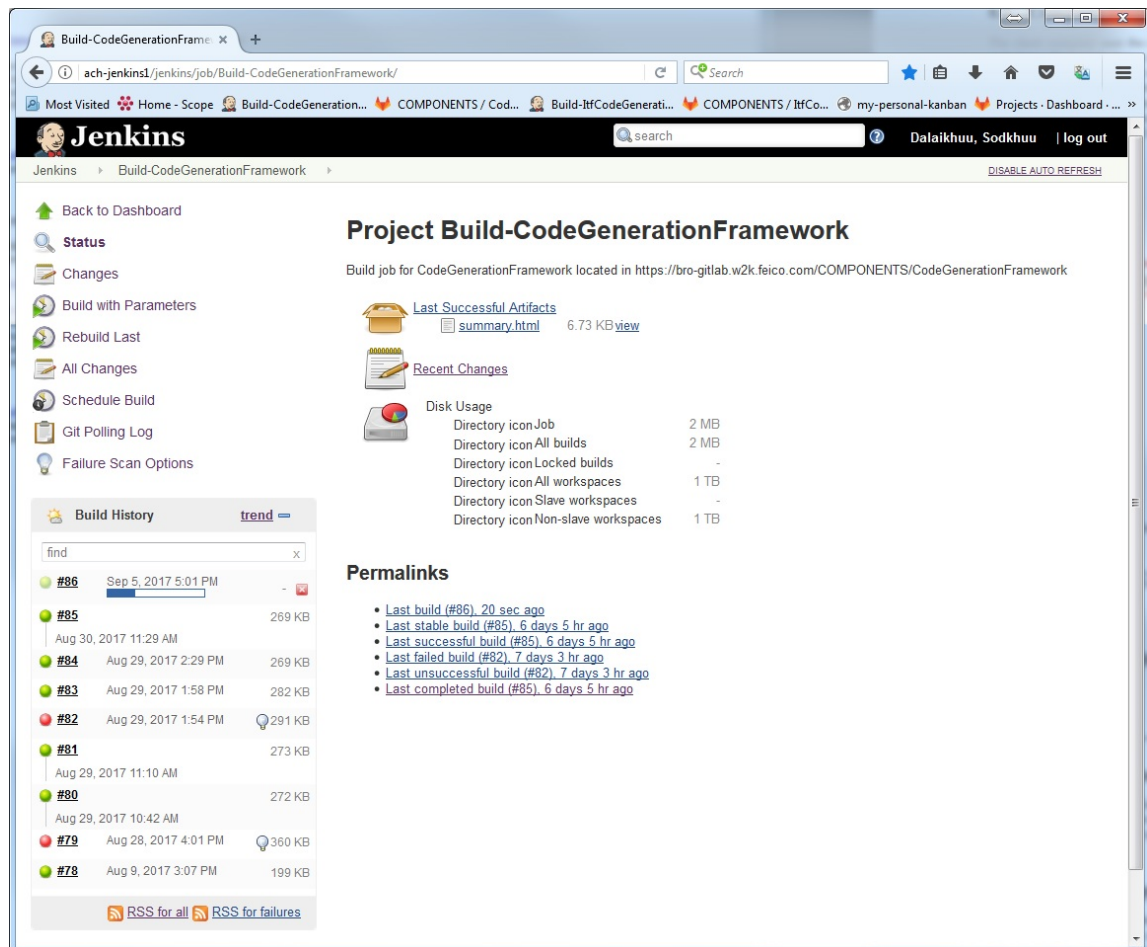


Figure 6.5: Jenkins build of the CodeGenerationFramework

6.3 Demo project component

A demo project component, *ItfCodeGenDemo*, has been created and deployed on the company built environment as a consumer of the *CodeGenerationFramework* component. Figure 6.6 shows COCA structure of the component. The component consists of the *CodeGenerationFramework* component as well as other company specific function group modules. The *ITF_CODE_GENERATION_DEMO* folder has five sub-folders such as *Objectmodel*, *sdk*, *shared*, *Test*, and *build*. The *Objectmodel* folder contains the demo project source code. The *sdk* contains shared files which is accessible among all other components. The *shared* is a build infrastructure specific folder. The *Test* folder contains unit tests of the demo project. The *build* folder contains client company development tool specific settings files.

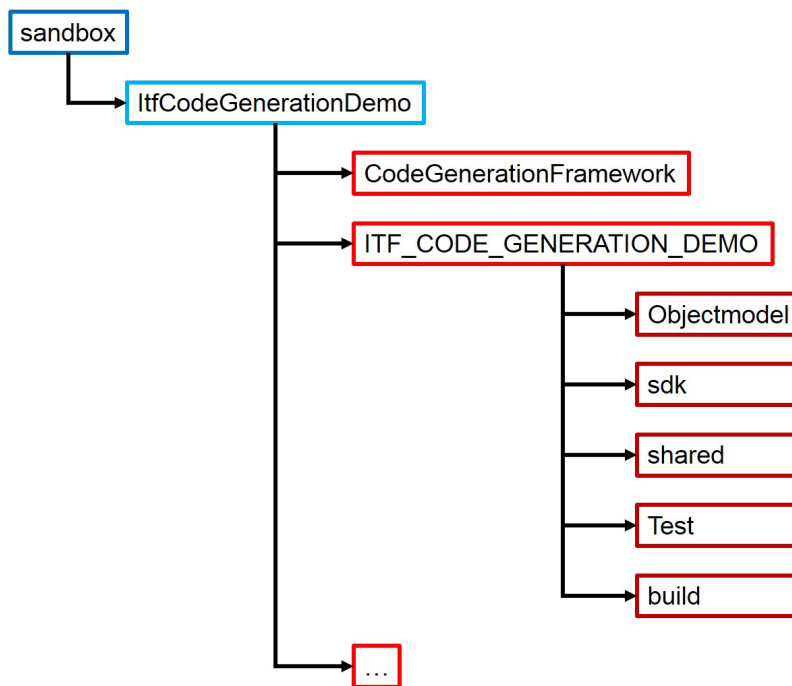


Figure 6.6: COCA structure of *ItfCodeGenDemo*

The demo project component gets the code generation framework component automatically using the company development tool. The build infrastructure synchronizes all dependencies for a component. The dependencies are configured in the **build** folder **component.xml** file. Sample configuration settings are as follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<Component>

  <Vitals>
    <Include Module = "*" />
  </Vitals>

  <Dependencies>
    <!-- Put dependencies of ITF_CODE_GENERATION_DEMO here -->
    <Component
      Name = "CODE_GENERATION_FRAMEWORK"
      Label = "CODE_GENERATION_FRAMEWORK_1.0.0_DEV27.2"
    />
  </Dependencies>

</Component>

```

The demo project structure is mimicked one of the existing platform in the built environment called Acquisition Server. Figure 6.7 shows package diagram of the demo project and its implementation in Visual Studio.

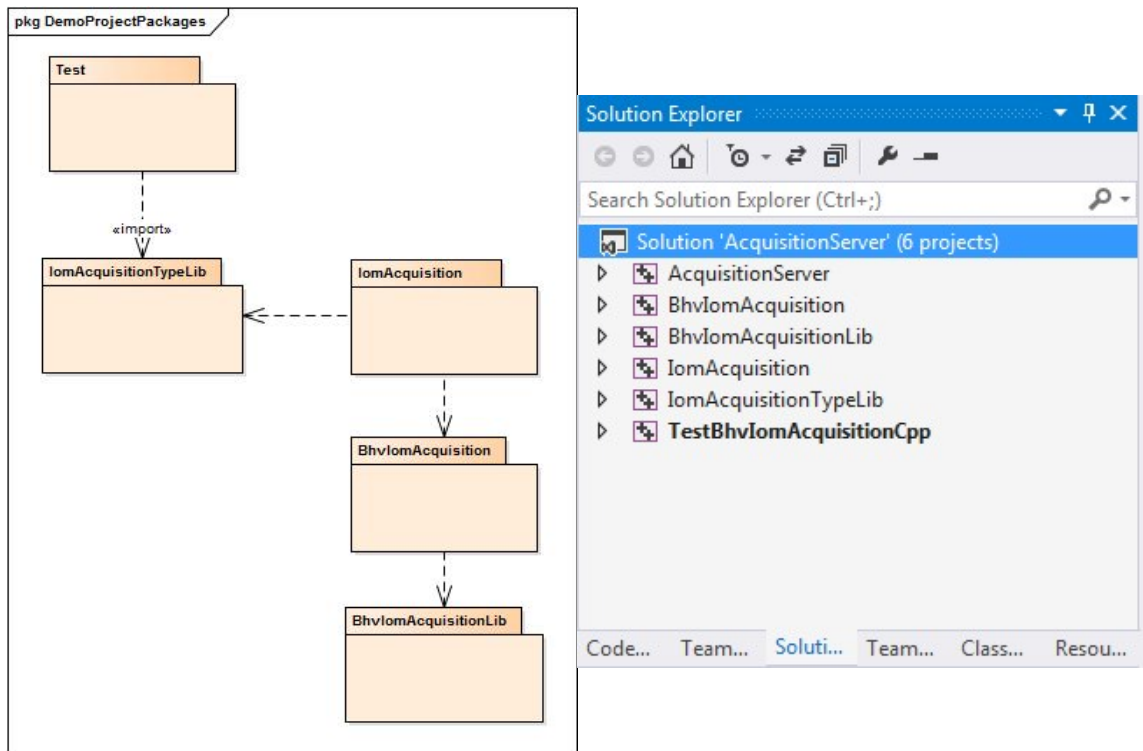


Figure 6.7: Package diagram of the demo project component

In general, the project consists of five projects representing three layers such as client application layer, COM layer, and server layer. The client application layer is represented by a test project, *TestBhvlomAcquisitionCpp*, which is implemented in google test framework. The COM layer consists of two sub-projects such as *IomAcquisition* and *IomAcquisitionTypeLib*. The first sub-project includes an IDL file and an header file that imports a type library generated from the IDL file. The second sub-project creates the COM environment around the type library. The server layer has the COM-to-C++ wrapper code as well as the C++ implementation in *BhvlomAcquisition* and *BhvlomAcquisitionLib* projects respectively.

Figure 6.8 shows class diagram of the demo project. It consist of COM classes, C++ interfaces and C++ implementations. The classes colored in red are the files that we generate using our framework. The classes colored in white are manually developed.

As same as the *CodeGenerationFramework* component, the demo project component uses Jenkins for build, deployment, and automation. The successful build results of the component are stored in the holding area as well. Figure 6.9 shows sample successful builds of the demo project which are stored in the holding area. The results are labeled using the same form as the *CodeGenerationFramework* component.

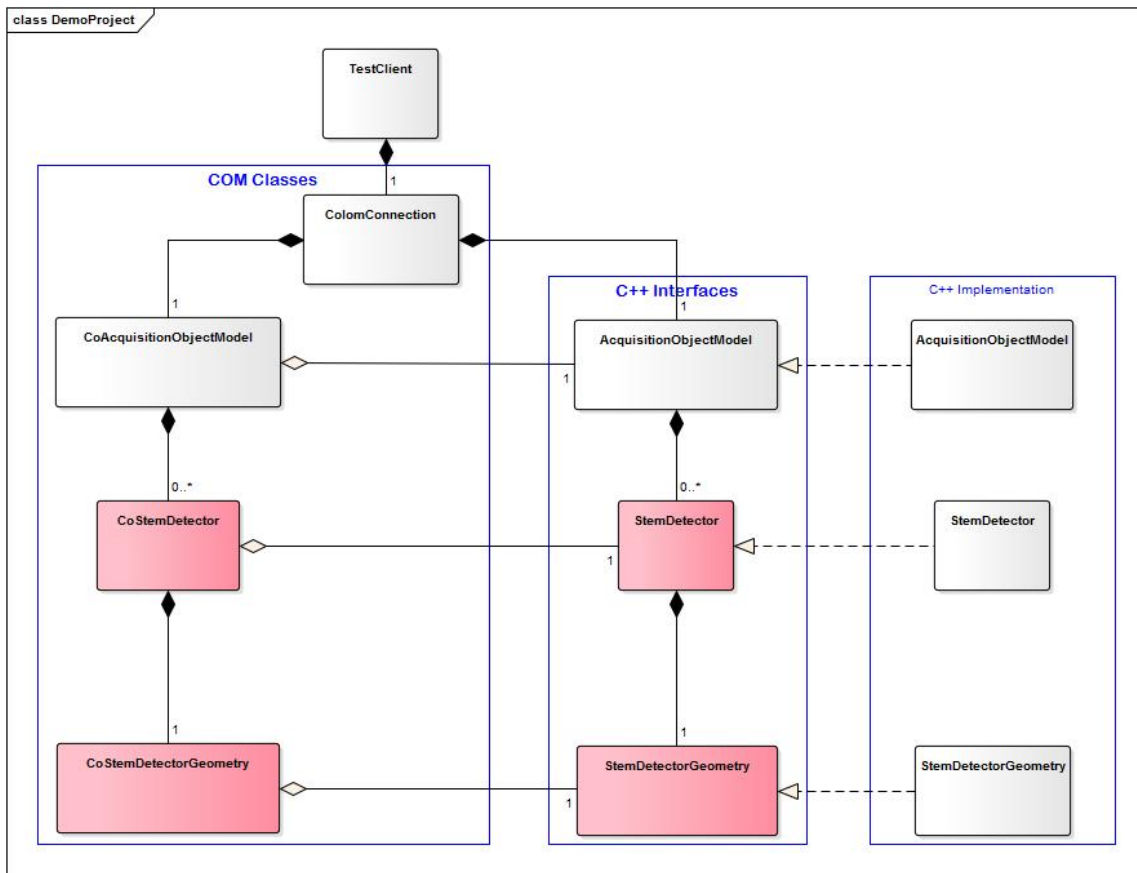


Figure 6.8: Class diagram of the demo project component

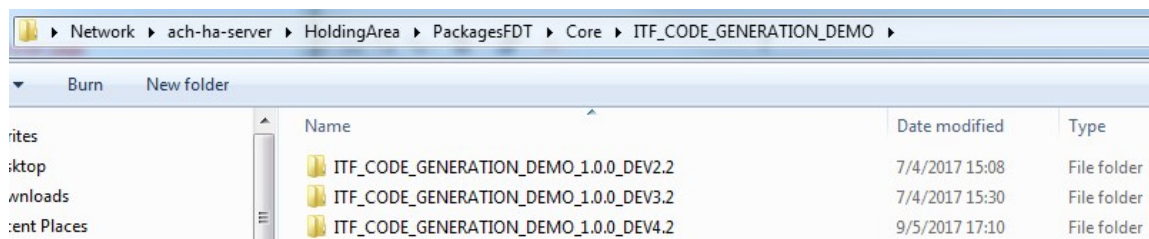


Figure 6.9: Jenkins build result stored in the Holding Area

Chapter 7

Verification and Validation

This chapter presents the process of verification and validation the framework. The strategy, test of the demo project, and trial with an end-user are introduced in the following sections.

7.1 Verification and validation strategy

The verification and validation processes determine whether the developed framework conform to the requirements of the project and whether the framework satisfies its intended application and user needs. Verification evaluates the framework to determine whether the generated code artifacts satisfies the existing test criteria. Validation evaluates the framework at the end of the development process to determine whether it satisfies specified requirements.

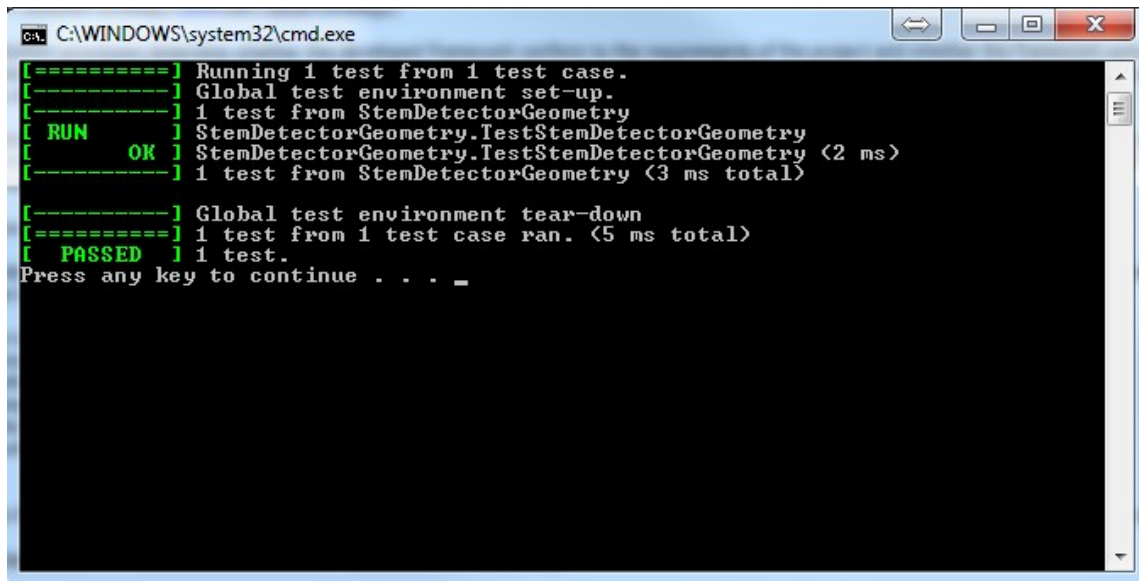
The verification and validation strategy was decided based on consultation with the company stakeholders. The strategy consists of two phases: test with a demo project (verification), and trial with an end-user (validation). The verification took place parallel with the framework implementation process and the validation took place in the final two months of the project. The next two sections explain each phases in more details.

7.2 Test of the demo project

The framework was continuously developed in parallel with the demo project. The demo project has a unit test for the sample component. The testing was performed during development to verify generated artifacts by following the below steps:

- Selecting an interface
- Collecting knowledge about the interface
- Defining methods in the interface
- Creating templates for the wrapper code artifact to be generated
- Compiling the file that contains the interface definition using the custom build tool (this action generates type libraries and code artifacts)
- Executing unit test

The demo project mimics the *StemDetectorGeometry* interface. Therefore, we have implemented an unit test in GoogleTest framework for the component inspired by existing unit test. Figure 7.1 shows unit test result of the stem detector geometry.



```
C:\WINDOWS\system32\cmd.exe
[====] Running 1 test from 1 test case.
[-----] Global test environment set-up.
[-----] 1 test from StemDetectorGeometry
[ RUN   ] StemDetectorGeometry.TestStemDetectorGeometry
[ OK    ] StemDetectorGeometry.TestStemDetectorGeometry (2 ms)
[-----] 1 test from StemDetectorGeometry (3 ms total)

[-----] Global test environment tear-down
[====] 1 test from 1 test case ran. (5 ms total)
[ PASSED ] 1 test.
Press any key to continue . . . _
```

Figure 7.1: Unit test result

The demo project has been configured as a compliant component in the company built environment, therefore, the test execution is included in the Jenkins build process. After building all projects of the component, the Jenkins builds all tests and if all tests pass the build is considered as successful. Figure 7.2 shows the Jenkins build console output log. This result shows that our framework has been successfully integrated in the company built environment and component build procedure.

```

ach-jenkins1/jenkins/job/Build-It Code Generation Demo/17/consoleText
Most Visited Home - Scope Build-CodeGeneration... COMPONENTS / Cod... Build-ItCodeGenerati... COMPONENTS / RfCo... my-personal-kanban Projects - Dashboard - ...
..\\FEI_CPPLIBS\\sdk\\include\\ /I..\\..\\COMMON_TYPES\\sdk\\include\\ /I..\\..\\IMAGING\\sdk\\include\\ /I..\\..\\IMAGING_TEM\\sdk\\include\\ /I..\\..\\INFRA\\sdk\\include\\ /I..\\..\\Libraries\\googlemock\\include\\ /I..\\..\\Libraries\\boost /I..\\..\\FEI_CPPLIBS\\sdk\\include\\gen\\ /I..\\..\\COMMON_TYPES\\sdk\\include\\gen\\ /I..\\..\\IMAGING\\sdk\\include\\gen\\ /I..\\..\\IMAGING_TEM\\sdk\\include\\gen\\ /I..\\..\\INFRA\\sdk\\include\\gen\\ /I..\\..\\Libraries\\googlemock\\include\\ /Zi /nologo /W4 /WX /sdl /Od /Oy- /D WIN32 /D DEBUG /D WINDOWS /D WIN32_WINNT=0x0601 /D _SCL_SECURE_NO_WARNINGS /D _UNICODE /D UNICODE /GF /Gm- /EHa /RTC1 /MDd /GS /Gy /fp:precise /Zc:wchar_t /Zc:forScope /GR /Yu"stdafx.h" /Fp"Debug\\TestBhvIomAcquisitionCpp.pch" /Fo"Debug\\ /Fd"Debug\\vc120.pdb" /Gd /TP /analyze- /errorReport:queue TestStemDetector.cpp
7/4/2017 3:29:59 PM: TestStemDetector.cpp
7/4/2017 3:30:00 PM: C:\Program Files (x86)\BullseyeCoverage\bin\link.exe /ERRORREPORT:QUEUE /OUT:"E:\SB\master_Build\bin\Debug\\TestBhvIomAcquisitionCpp.exe" /INCREMENTAL:NO /NOLOGO /LIBPATH:..\\shared\\lib\\Debug\\ /LIBPATH:..\\..\\shared\\lib\\Debug\\ /LIBPATH:..\\..\\sdk\\lib\\Debug\\ /LIBPATH:..\\..\\Libraries\\boost\\lib\\ /LIBPATH:..\\..\\FEI_CPPLIBS\\sdk\\lib\\Debug\\ /LIBPATH:..\\..\\COMMON_TYPES\\sdk\\lib\\Debug\\ /LIBPATH:..\\..\\IMAGING\\sdk\\lib\\Debug\\ /LIBPATH:..\\..\\IMAGING_TEM\\sdk\\lib\\Debug\\ /LIBPATH:..\\..\\INFRA\\sdk\\lib\\Debug\\ /LIBPATH:..\\..\\Libraries\\googlemock\\lib\\Debug\\ kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib odbccp32.lib odbccp32.lib /MANIFEST /MANIFESTUAC:"level='asInvoker' uiAccess='false'" /manifest:embed /DEBUG /FDB:"E:\SB\master_Build\bin\Debug\\TestBhvIomAcquisitionCpp.pdb" /SUBSYSTEM:CONSOLE /TLBID:1 /DYNAMICBASE /FIXED:NO /NXCOMPAT /IMPLIB:"E:\SB\master_Build\bin\Debug\\TestBhvIomAcquisitionCpp.lib" /MACHINE:X86 /MACHINE:I386 Debug\\stdafx.obj
Debug\\TestStemDetector.obj
7/4/2017 3:30:00 PM: Creating library E:\SB\master_Build\bin\Debug\\TestBhvIomAcquisitionCpp.lib and object E:\SB\master_Build\bin\Debug\\TestBhvIomAcquisitionCpp.exp
7/4/2017 3:30:00 PM: gmock.lib(gtest-all.obj): warning LNK4099: PDB 'vc120.pdb' was not found with 'gmock.lib(gtest-all.obj)' or at 'E:\SB\master_Build\bin\Debug\\vc120.pdb'; linking object as if no debug info
7/4/2017 3:30:00 PM: TestBhvIomAcquisitionCpp.vcxproj -> E:\SB\master_Build\bin\Debug\\TestBhvIomAcquisitionCpp.exe
7/4/2017 3:30:00 PM: "E:\SB\master_Build\bin\Debug\\TestBhvIomAcquisitionCpp.exe" --gtest_output=xml:E:\SB\master_Build\bin\Debug\\TestBhvIomAcquisitionCpp_gtest.xml
VCEnd
7/4/2017 3:30:00 PM: [=====] Running 1 test from 1 test case.
7/4/2017 3:30:00 PM: [-----] Global test environment set-up.
7/4/2017 3:30:00 PM: [-----] 1 test from StemDetectorGeometry
7/4/2017 3:30:00 PM: [ RUN ] StemDetectorGeometry.TestStemDetectorGeometry
7/4/2017 3:30:00 PM: [ OK ] StemDetectorGeometry.TestStemDetectorGeometry (1 ms)
7/4/2017 3:30:00 PM: [-----] 1 test from StemDetectorGeometry (1 ms total)
7/4/2017 3:30:00 PM: [-----] Global test environment tear-down
7/4/2017 3:30:00 PM: [=====] 1 test from 1 test case ran. (2 ms total)
7/4/2017 3:30:00 PM: [ PASSED ] 1 test
7/4/2017 3:30:00 PM: Deleting file "Debug\\TestBhvI.CE5B7494.tlog\\unsuccessfulbuild".
7/4/2017 3:30:00 PM: Touching "Debug\\TestBhvI.CE5B7494.tlog\\TestBhvIomAcquisitionCpp.lastbuildstate".
7/4/2017 3:30:00 PM: Done building project "E:\SB\master_Build\ITF_CODE_GENERATION_DEMO\\TestBhvIomAcquisitionCpp.vcxproj".
7/4/2017 3:30:00 PM: Build SUCCEEDED.
7/4/2017 3:30:00 PM: 2 Warning(s)
7/4/2017 3:30:00 PM: 0 Error(s)
7/4/2017 3:30:00 PM: Time Elapsed 00:00:03.008
7/4/2017 3:30:00 PM: Module test\TestBhvIomAcquisitionCpp Debug\Win32 build finished
=====
7/4/2017 3:30:00 PM: Finished Unidentified_03b998e87de14acea4dc79ff18ff7bfe build
Time Elapsed 00:00:23.955
test
Highlight All Match Case Whole Words 12 of 32 matches

```

Figure 7.2: Jenkins test result

7.3 Trial with end-user

The trial with an end-user was centered on parsing existing IDL files and generating existing manually developed codes. During this process, an interface developer has used our framework to generate COM-to-C++ wrapper code artifacts based on the existing IDL files. The interface developer has used his own workstation and responsible code base. He has modified his component configuration to include our code generation framework as dependency.

7.3.1 Parsing IDL files

The first challenge was to parse sample IDL files selected by end-user from existing code base. This activity was focused on validating the IDL grammar that we have implemented. The validation was performed using the Irony Grammar Explorer tool. The following existing IDL files are used for testing the parser:

- ImageRotation.idl
- IOptics.idl
- IOpticsDoseData.idl
- ipeo_column.idl
- IItemFocus.idl

All files are successfully parsed conforming to our IDL grammar. Figure 7.3 shows parsed result of the IOptics.idl file as an example. The other parsed results are included in Appendix D.

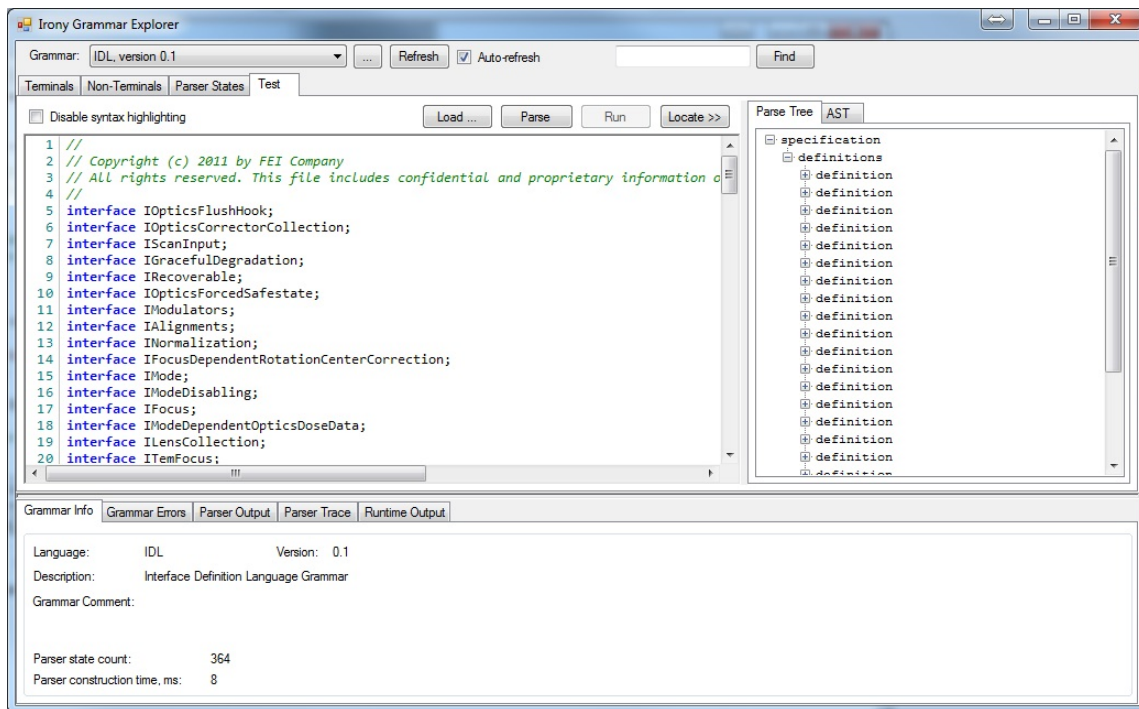


Figure 7.3: Parsed result of the IOptics.idl file

7.3.2 Generating code artifacts

The second challenge was to generate existing manually developed code artifacts. In this activity, we used the wrapper code of the IDL files that we have used in the previous activity. The wrapper code files are as follows:

- ImageRotation.idl
 - ComImageRotation.cpp
 - ComImageRotation.h
- IOptics.idl
 - ComOptics.cpp
 - ComOptics.h
- IOpticsDoseData.idl
 - ComOpticsDoseData.cpp
 - ComOpticsDoseData.h
- IItemFocus.idl
 - TemFocusProxy.cpp
 - TemFocusProxy.h

We have created templates for each sample wrapper code using transformed interface model information and generated visually as same code artifacts as the existing manually developed code files.

7.3.3 Feedback from the end-user

After the validation session with end-user, we have asked for feedback concerning his impression about using the framework. The questions are based on the ease-of-use requirement as follows:

- Q1: How much time required to learn the tool?
Not much, just followed the instructions.
- Q2: How much time required to get used to the tool?
I haven't had the time to get used to the tool, because I only helped to prove the concept. But as a user, I would say that there are not much to get used to.
- Q3: How much time required for generating code artifacts?
Initially this step took some time, but this was because of the infrastructure to select the component. This is not related to the tool itself.
- Q4: How much time required to learn the process of generating code artifacts?
As soon as the tool is in place for a component, it only consists of repeating the same steps for different IDL files.
- Q5: How easy was it to associate with the domain concepts?
I don't know what you mean. How easy it is to relate the generated artifacts to the input? Or get used to the terms used in the code generation framework? In any case, the 15 (or so) minute presentation was enough to understand the goal of the tool, the general technical approach and the artifacts it generates.

7.4 Proposal for COM-to-C++ wrapper code generation

During the trial with end-user, we have faced several obstacles for generating wrapper code from IDL file. For instance, there are no current separation of COM and C++ layers in Optics team. The behaviors of all components are implemented directly in the COM layer. On one hand, it is difficult to create a generic template for such code. On the other hand, creating templates for every exceptional case would be worse than implementing them by hand. Hence, based on collaboration and feedback from the end-user, we have proposed a generic template for COM-to-C++ wrapper code generation. The proposed templates are included in Appendix E

We have used the generic template to generate one of the existing wrapper code for Energy Filter Service because the implementation of its wrapper code is well separated into COM and C++ layers. Therefore, it is considered as a good example for the generic template and we suggest to start using the framework in this component. However, there has been on-going changes in the code base of the Energy Filter Service at the same time our validation activity. Thus, we could not generate the actual production wrapper code for this component. Fortunately, the Energy Filter Service component itself is relatively new and under development for further improvement. Hence, we suggest the Thermo Fisher Scientific company to adopt our code generation framework for software development starting from the Energy Filter Service component. Furthermore, the end-users are enthusiastic for using our framework and they are suggesting to adopt of the framework in the road-map of the Optics team as well for using the code generation after editing the current Optics code base into clearly separated COM and C++ layers.

Chapter 8

Conclusions

This chapter focuses on conclusions of this project, elaborating achieved results and added values to the stakeholders. It also presents future work and alternatives for improving the framework.

8.1 Results

Model-based code generation is a set of ideas that aim for faster and cheaper artifact creation by automating repetitive tasks that are derivable from high-level models. Implementing and deploying these ideas into the Thermo Fisher Scientific software development process is the focus of this project.

The framework is based on the Model-Driven Architecture (MDA). Model transformation is the corner stone of this architecture. In our case, transformation from Domain Specific Language (DSL) to code artifacts is considered as a large abstraction gap because source and target models are in different technical spaces such as Extended Backus-Naur Form (EBNF) and Meta Object Facility (MOF). Therefore, we have introduced model-to-model transformation activity before model-to-code transformation activity. This design decision has led us to construct the framework with three components such as parser, model-to-model transformer, and model-to-code transformer. The parser component parses the input DSL model to an Abstract Syntax Tree (AST). The model-to-model transformer transforms the AST to an object-oriented interface model. The model-to-code transformer transforms the interface model to code artifacts.

We have chosen the Interface Definition Language (IDL) as the DSL in the project scope. The client company has been using this language for interface models for decades combining with the Microsoft Component Object Model (COM) technology. Thus, it was natural decision for us to use this language as the DSL for our framework. The Irony .NET Language Implementation Kit was used for implementing the IDL grammar. The tool supports the EBNF like notation for defining grammar rule. Thus, our IDL grammar implementation, which is based on the OMG IDL grammar, has followed the EBNF for defining its rule.

We have designed an interface metamodel for the framework based on the MOF. The metamodel structure combines general object-oriented programming language concept and COM. Further, it is extensible and easily modifiable for further demands.

The parser component of the framework is developed using the Irony .NET Language Implementation Kit. This kit is suitable for C# and .NET based environment comparing to other existing technologies, further, directly suitable for client company's software development environment. Therefore, this kit fulfills the necessity of having an usable tool for interface developers.

The model-to-model transformer component of the framework is developed using a structure driven approach. Currently, there are no proven technology exists for .NET framework so that we have decided to implement the component in C# environment by using direct mapping approach from source to target model.

The model-to-code transformer component of the framework is developed using Microsoft T4. This technology fulfills the ease-of-use requirement because it includes human-readable output artifact definitions. Also, the technology is developed by Microsoft which means it is suitable for C# and .NET framework.

In order to integrate our framework with the current work-flow of software development process of the client company, we have configured a custom build tool for IDL file compile action in Visual Studio. The custom build tool combines the Microsoft IDL (MIDL) compiler, which is used for generating type libraries, with our code generation framework tool.

The framework is deployed on the client company's built environment as a component. The deployed component is available for all developers of the company through the company build infrastructure.

Furthermore, the framework is verified and validated using unit test and end-user respectively. For verification, we have created a unit test inspired by existing unit test. For validation, we have performed a trial run with an end-user. During the trial run, we have parser various IDL files using our grammar and generated several code artifacts that are visually the same. However, due to existing code base changes of the company, we could verify the generated artifacts using smoke test and integrate to the production code base. Yet, we have created a proposal for ideal wrapper code generation template based on the input of the end-users. The template could be good starting point for using the framework and standardizing the wrapper code structure in the client company.

In conclusion, during the lifetime of the project, we have successfully designed, implemented, and deployed a generic framework which generates code artifacts based on input DSL files. The framework is easy to use with the custom build tool. Also, the framework is available for the client company's developers due to successful deployment of the code generation framework component. Further, the framework is extensible due to its component based nature.

8.2 Design Opportunities Revisited

The project was aimed at fulfilling three design opportunities (non-functional requirements) such as extensibility, ease-of-use, and reliability. This section summarizes the results of each implementations.

- **Extensibility**

The framework was designed the extensibility in mind from the beginning. We have used the factory method design pattern to allow future extension for supporting different grammar, model-to-model transformer, and model-to-code transformer. We have used an open source library for the parser component as well as implementing the grammar. The component based structure allows easy change for integrating or replacing a better technology. The template based technology is considered as mature because it is supported by Microsoft and is under constant maintenance. In conclusion, the framework is extensible.

- **Ease-of-use**

The end-user has evaluated the framework at the end of the project. They have indicated

that the tool is easy to learn by just following the instructions and a 15 minute presentation was enough to understand the goal, general technical approach, and generated artifacts of the tool. The tool is easy-to-use.

- **Reliability**

The parser has successfully parsed various sample IDL files into parse tree. Also, the template based technology allows the interface developers to design templates that suits their needs. The model-to-model transformer transforms the parse tree information to interface model information by following simple mapping approach. Therefore, the output code artifacts are reliable for using instead of manually written code artifacts.

8.3 Future work

During the development of the project, we have identified future possibilities and improvements as well as features that were not implemented due to various constraints such as time, technology, complexity, and added value. The ideas for future work are listed below:

- We have implemented the IDL grammar with focus of generating wrapper code. However, there might be some parts that are not covered with the grammar but could be useful for further code generation projects. Therefore, it is important to improve the grammar in the future for further demands of various code generation projects.
- The model-to-model transformer works for simple use case of generating COM-to-C++ wrapper code based on simple IDL models. Thus, it would need further extension to fully cover all type of IDL models and wrapper code that the client company use.
- The code generation framework currently supports primitive type transformations from source to target model. Classifier type conversions are purely based on the identifier of the parameter type. Therefore, this part needs some work to be done. The transformer might need to check if the classifier is defined in the current IDL file or in one of the imported IDL files.
- In order to make use of another IDL file definitions, the framework would need to parse other type libraries or IDL files as input as well. There are common use of importing another type libraries or IDL files in an IDL file and using the type definitions from those files.
- Interfaces and enumerations are used as type in IDL model as well. Therefore, these should be implemented in the future.
- We have proposed an ideal template for generation COM-to-C++ wrapper code. However, this template needs to be verified and modified for more generic use for all interfaces of the client company.
- Client side wrapper code generation is as important as server side wrapper code generation. The client company wants to create clear separation of COM and C++ layers on the client side of the software as well. Unfortunately, during the lifetime of this project, we could not find existing good example for the client side wrapper code and could not propose ideas due to time limitation. Yet, it is possible to extend the framework for generating such wrapper code.

Chapter 9

Project Management

This chapter describes the project management process that was conducted during the life-time of the project.

9.1 Introduction

The project is more practical than theoretical which means we have developed a prototype of usable and extensible framework. We have decided to follow an iterative-incremental development combined with an agile methodology. By following this approach, we have held two meetings per week with the project mentor for progress analysis, requirement refinement, and planning.

Iterations usually lasted one month. The outputs of the iterations were an updated version of the prototype that implements the refined requirements. The requirements were evaluated and redefined or discarded for following iterations.

9.2 Work-Breakdown Structure (WBS)

Figure 9.1 shows work-breakdown structure. The work-breakdown structure is the way in which the project had been decomposed into small packages with deliverable oriented components. The project is divided into five main activities as follows:

- **Domain research:** Research and study the current code base of the client company including interface technology and wrapper code implementations, as well as research of the Model-Driven Architecture
- **Technology research:** Research and implementation of the technologies that we used in the framework
- **Design and implementation:** Usage of the results of the previous two activities in order to build a framework for our project
- **Deployment:** Deployment of the framework and the demo project on the company built environment
- **Verification and Validation:** Test of the generated code artifacts and trial run with an end-user
- **Documentation:** Redaction of the expected documentation deliverables

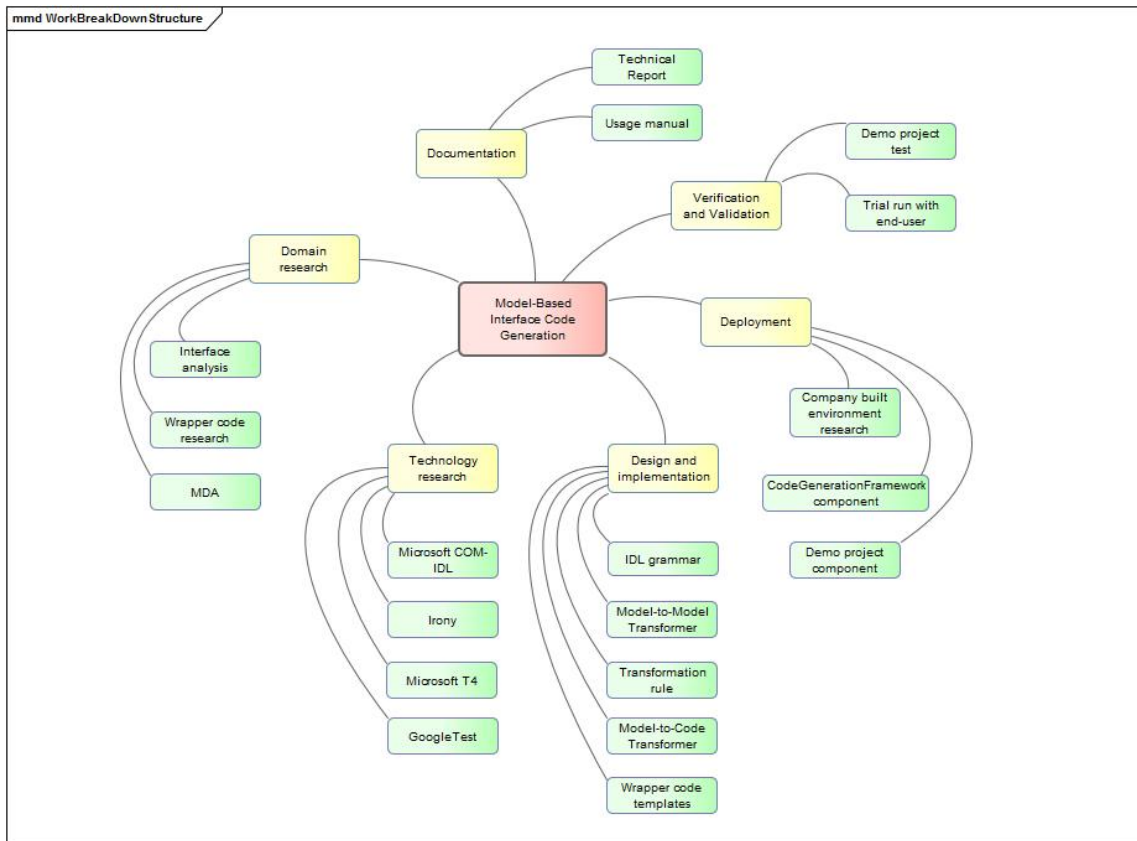


Figure 9.1: Work-breakdown structure

9.3 Project planning

This project was executed in nine months, starting in January 2017 until the end of September 2017. The duration of nine months consists of 5 weeks spent on university events and vacation, and 34 entire working weeks at the client company. Figure 9.2 shows general plan overview containing chronological sequence and duration of high-level activities carried during this project. From this plan, we distinguish three main project phases that match with the five main activities of the WBS:

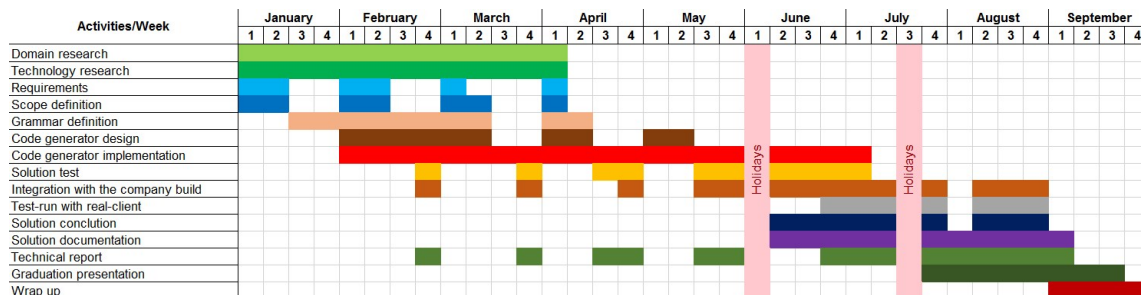


Figure 9.2: Project plan

- **Research:** Initial phase covering the first three months of the project in which domain and technology research were the main activities
- **Design and implementation:** Intermediate phase covering five and a half months. The incremental-iterative development approach was applied and the prototype was developed

- **Closure:** Final phase covering one and a half month at the end of the project period in which the project was concluded and the final documentation was delivered

As Figure 9.2 shows, during the intermediate phase, relevant activities were split and overlapped. This describes iterations life-cycles in which more requirements were implemented into the prototype.

9.4 Feasibility analysis

The feasibility analysis was performed after understanding the problem and domain. We have identified existing or possible challenges, issues, and risks of the project.

9.4.1 Challenges

Several challenges were encountered for the project. This section discusses these main challenges and the strategies that we took into account to address them.

Scoping the project

Scope definition is essential for a project. The scope and deliverables define the acceptance criteria of a project. The project scope would be defined using extended discussion with all stakeholders where we gather their concerns and prioritize deliverables. The project has time limitation of nine months, therefore, the deliverables should be reasonable to be implemented in the time-frame.

Project deployment

Deploying the framework on the company built environment is crucial for this project. The framework should be available for all developers who wants to generate wrapper code for the components that they work on. Therefore, this activity would demand certain understandings about structure of the built environment, procedure of adding new component, configuration of a proper component, and proper validation of using the code generation component on another consumer component. Additionally, inserting a new element in a development process has to be considered carefully. The added element has to be easy to understand by developers and its benefits and advantages should be defined in terms of saved time of the user.

Automation of repetitive task

Automation of repetitive task is a key goal of this project. The repetitive tasks are tedious and mechanical, as well as error prone as a consequence. These activities contemplate wrapper code of COM interfaces. All the interface wrapper code contain similar parts following certain patterns in the high-level view. These parts are repeated and applied to different elements. However, having a pattern does not ensure that all parts for creating wrapper code can be automated. There are parts that cannot be related to a predefined pattern depending on technology and system features. Thus, the necessity of defining automated and non-automated parts have emerged. The information source for generating code artifacts was the domain specific language that we have chosen. All automated parts are based on these informations and the parts that are not modeled in the domain specific language were identified as non-automated parts.

9.4.2 Risks

A number of risks were identified during the project. This section discusses these risks, their possible impacts on the project, and corresponding mitigation strategy from project management perspective.

Unavailability of ideal wrapper code

Finding an ideal sample wrapper code for code generation might not be straightforward activity. Any legacy code base might include various different dialects of implementation style depending on developers and teams due to manual implementation and lack of standardization.

The impact of this risk is that it might result less generic template creation in the framework. Moreover, additional time and domain knowledge would be necessary for creating a generic template for automatic code generation that is suitable for all interface developers.

The mitigation strategy that we have taken on this risk includes two activities. First, we have decided to choose the best and simplest example wrapper code from the current code base based on the consultation of interface developers. These wrapper code implements the separation of COM and C++ layers. Second, we have decided to propose an ideal wrapper code template for separating COM and C++ layers.

Fulfilling requirements in limited time-frame

Some of requirements might not be fulfilled in the time-frame of this project due to unpredicted obstacles.

The impact of this risk is that the project might not meet all requirements identified from stakeholders.

The mitigation strategy that we have taken on this risk was that prioritizing requirements together with the stakeholders. The requirements that has *should* or *optional* priority could be carried on as a future work. The project plan and progress would be revised continuously, and updated the list of requirements and their priorities. Furthermore, the framework would be designed extensible and easy-to-change for future improvements.

Stakeholder unavailability

Stakeholders might be unavailable due to several excuses such as illness, holidays, and work load. In any of these circumstances, the project should proceed with sufficient inputs and feedbacks.

The impact of this risk is that some stakeholders might not be available for detailed discussions. Thus, the project might lack inputs and feedback on particular activities which might result wrong decisions, priorities, and unnecessary time consumption.

The mitigation strategy that we have taken on this risk was that prioritizing the importance of inputs and feedbacks from the stakeholders, create plans for meetings in advance, and always try to have a backup strategy. An example backup strategy was that in case an important stakeholder is unavailable, try to gather insights from alternate stakeholders.

9.5 Project execution

Requirement specification, technology research as well as design and implementation had been executed simultaneously during this project. This approach aims to validate technology research through implementation as soon as possible. This implementation have produced a prototype that demonstrated our results, and provided input to future refinements. The prototype was evolved over time to become the final solution.

In accordance with the PDEng project guidelines, the Project Steering Group (PSG) meeting was held in which project progress was presented by the PDEng candidate. The members of the

PSG are university supervisor, company supervisor, and company mentor. The main topics of these meetings were the project status together with a demonstration of the prototype. The goal of these meetings was informing, discussing, and getting feedback from the PSG members. These feedback sessions had been working as a method of validating and keeping track of the project direction.

The iterative-incremental approach and the prototype development have allowed stakeholders to have a clear view of the project progress, technology features and potential, as well as requirement implementation. Additionally, it helped to detect risks and to make decisions to address them.

Chapter 10

Project Retrospective

This chapter finalizes the document by providing a reflection on the project based on the author's perspective. It also depicts the revision of the design opportunities, defined in the beginning of the project.

10.1 Reflection

During these nine months, my journey has been challenging but rewarding. Due to its real industrial setup, the project has enhanced my professional and personal skills by presenting challenges from technical to managerial perspectives. These challenges has led me to learn and apply best practices and identify my strengths and improvement points.

At the beginning of the project, I had to learn various new technologies, related works, and concepts. I spent first three months on domain and technology research. This phase included not only reading activities but also prototyping examples. Thus, I have gathered sufficient experience on the relevant technologies and related works in the short time.

Scoping the project was crucial as any other industrial project. The project serves for two purposes such as PDEng thesis and industry driven technical project. Therefore, I had to define clear scope for managing the limited time to deliver customer demands as well as university standard. Periodic and structured discussions with the stakeholders have proven to be useful to keep everybody on the same page.

In the design of the framework, I have used three design patterns such as factory method, builder, and template method. During the design and implementation phase, I was constantly trying to apply design patterns in order to make the framework extensible, organized, easy-to-understand, and maintainable. Therefore, in most part of the framework design, I have iteratively analyzed the code and applied the design patterns accordingly.

I had my first meetings with the end-users in the second half of the project. The meetings have provided me broader perspective on the problem and helped me to understand the company vision towards code generation. With that in mind, I filtered a rule of thumb that is if you want to obtain broader overview about the project then meet the actual problem holders as early as possible. This activity helps to observe the important aspects of the project and act accordingly.

Managing individual project has been challenging. My previous work experience was based on team oriented projects, thus, I had to learn self management approaches and best practices. I have studied couple of previous PDEng project reports and discussed with my fellow PDEng trainees to learn from their approaches. This method has helped me to create my own approach

which was suitable for this project.

Overall, I have had a challenging and beneficial experience with this project. Practicing software development, cooperating with technical and managerial people, and managing their expectations have improved my both hard and soft skills. I have grown professionally with these experiences. Moreover, use of new technologies and application of different techniques has broadened my overview and opened promising outlook for the future.

Glossary

10.2 Abbreviations

PIM	Platform Independent Model
PSM	Platform Specific Model
MDA	Model-Driven Architecture
COM	Component Object Model
IDL	Interface Definition Language
TEM	Transmission Electron Microscope
T4	Text Template Transformation Tool
PDEng	Professional Doctorate in Engineering
DSL	Domain Specific Language
EBNF	Extended Backus-Naur Form
OMG	Object Management Group
MOF	Meta Object Facility
GUID	Globally Unique Identifier
API	Application Programming Interface
MIDL	Microsoft Interface Definition Language
UML	Unified Modeling Language
CAFCR	Customer, Application, Functional, Conceptual, and Realization
TLB	Type library
FIDL	FEI Interface Definition Language
MBIF	Model-Based Interface Framework
AST	Abstract Syntax Tree

Bibliography

- [1] J. Miller, Joaquin; Mukerji, *MDA Guide Version 1.0.1*, Object Management Group, 2003. 2
- [2] J. Bézivin, “On the unification power of models,” *Software & Systems Modeling*, vol. 4, no. 2, pp. 171–188, May 2005. [Online]. Available: <https://doi.org/10.1007/s10270-005-0079-0> 2, 12
- [3] —, “From object composition to model transformation with the mda,” in *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, ser. TOOLS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 350–. [Online]. Available: <http://dl.acm.org/citation.cfm?id=882501.884684> 2, 3, 20
- [4] T. Mens and P. V. Gorp, “A taxonomy of model transformation,” *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125 – 142, 2006, proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005). [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S1571066106001435> 3
- [5] H. Reimer, L.; Kohl, *Transmission Electron Microscopy*. Springer Science+Business Media, 2008. 4
- [6] A. Martinez, “Model based interface framework,” Eindhoven University of Technology, Tech. Rep., 2016. 4, 15
- [7] C. Xia, S. Prof, M. R. Lyu, M. Prof, K. fai Wong, and P. A. Fu, “Component-based software engineering: Technologies, quality assurance schemes, and risk analysis tools,” in *Seventh Asia-Pacific Software Engineering Conference*. 5
- [8] Microsoft. Component object model (com). [Online]. Available: [https://msdn.microsoft.com/nl-nl/library/windows/desktop/ms694505\(v=vs.85\).aspx](https://msdn.microsoft.com/nl-nl/library/windows/desktop/ms694505(v=vs.85).aspx) 5
- [9] W. Bilderbeek, H. Broeders, and A. van Rooijen, “Applying and extending the observer pattern in thrsim11,” 1998. [Online]. Available: <http://www.hc11.demon.nl/thrsim11/ddj/article.htm> 5
- [10] Microsoft. The interface definition language (idl) file. [Online]. Available: [https://msdn.microsoft.com/en-us/library/windows/desktop/aa378712\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa378712(v=vs.85).aspx) 5
- [11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. 7
- [12] G. Muller, “Cafcr: A multi-view method for embedded systems architecting; balancing genericity and specificity,” Ph.D. dissertation, Technische Universiteit Delft, 2004. 8
- [13] Microsoft. Com, dcom, and type libraries. 11
- [14] L. M. Garshol. Bnf and ebnf: What are they and how do they work? [Online]. Available: <http://www.garshol.priv.no/download/text/bnf.html> 20

- [15] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," 2003. 20
- [16] R. Ivantsov. (2013) Irony - .net language implementation kit. [Online]. Available: <https://irony.codeplex.com/> 22
- [17] —. (2008) Irony - .net compiler construction kit. [Online]. Available: <https://www.codeproject.com/Articles/22650/Irony-NET-Compiler-Construction-Kit> 22
- [18] F. Erata, M. Challenger, and G. Kardas, "Review of model-to-model transformation approaches and technologies," 2015. 23
- [19] Microsoft. Code generation and t4 text templates. [Online]. Available: <https://msdn.microsoft.com/en-us/library/bb126445.aspx> 23
- [20] OMG, *OMG Interface Definition Language Specification version 4.1*, 2017. [Online]. Available: <http://www.omg.org/spec/IDL/4.1/> 28
- [21] M. Piefel, "A common metamodel for code generation," in *CITSA 2006*, 2006, pp. 118–123. 30
- [22] Microsoft. (2017, 08) Idl attributes. [Online]. Available: [https://msdn.microsoft.com/nl-nl/library/windows/desktop/aa367042\(v=vs.85\).aspx](https://msdn.microsoft.com/nl-nl/library/windows/desktop/aa367042(v=vs.85).aspx) 34
- [23] —. (2017, 08) Writing a t4 text template. [Online]. Available: <https://msdn.microsoft.com/en-us/library/bb126478.aspx> 45
- [24] —. (2017, 09) Specifying custom build tools. [Online]. Available: <https://msdn.microsoft.com/en-us/library/hefydhhy.aspx> 46

Appendices

Appendix A

Appendix 1 Stakeholder Analysis

This appendix presents the stakeholders of this project, their interests and goals. The Thermo Fisher Scientific Company and the Eindhoven University of Technology are the involved parties and each of them has specific interests towards the project. The following sections introduce the stakeholders and present each party in detail.

A.1 Thermo Fisher Scientific

Thermo Fisher Scientific as a company is the owner and initiator of the project. They represent the source of knowledge, requirements, and expectations for this work.

STAKEHOLDER	High-Level Project Management
ROLE	Project Manager <ul style="list-style-type: none">The project manager is the person who defines the project goals, funds the project, and looks for the business value.
REPRESENTATIVE	Martijn Kabel / Arjen Klomp (Starting from April 1, 2017)
RESPONSIBILITIES	Monitor the project progress and deliverables. Plan the project future and deployment.
ACCEPTANCE (SUCCESS) CRITERIA	Same as Company supervisor's acceptance criteria
INVOLVEMENT	The trainee presents the project and results to the project manager one or two times in the final phase of the project.

STAKEHOLDER	Company Supervisor
ROLE	<p>Project Owner and Project Mentor</p> <ul style="list-style-type: none"> • The project owner is the person who organizes the project and has to make sure that the project goals (defined by the manager) are met on time and within the budget. • The project mentor is the person who decomposes the project goal, monitors the project progress daily and assist the trainee in the project related technical matters.
REPRESENTATIVE	<p>Project Owner: Andrei Radulescu Project Mentor: Erwin de Groot</p>
RESPONSIBILITIES	<ul style="list-style-type: none"> • Monitor, evaluate, assess, and provide regular feedback on the project progress and deliverables • Provide relevant domain knowledge, references, and contacts • Provide relevant information regarding the needs and requirements of the project • Evaluate and assess the robustness of the system, the functionality provided, and whether the solution meets the requirements • Review the final project report
ACCEPTANCE (SUCCESS) CRITERIA	<ul style="list-style-type: none"> • Timely report of the project • Project deliverable • Project deployment in the built environment of the company
INVOLVEMENT	<p>During the entire project by continuous communication via daily meetings on ad-hoc basis, regular weekly progress update meetings, and regular monthly project steering group meetings.</p>

STAKEHOLDER	Knowledge source and potential users
ROLE	Domain knowledge source
REPRESENTATIVE	Aldo Martinez David van Luijk Ronnie Smets Acquisition Server Team
RESPONSIBILITIES	<ul style="list-style-type: none"> • Provide relevant domain knowledge, references, and skills • Provide sample source materials and use-cases • Perform a test run using the prototype
ACCEPTANCE (SUCCESS) CRITERIA	N/A
INVOLVEMENT	During the entire project by continuous communication via meetings and e-mail on ad-hoc basis.

A.2 Eindhoven University of Technology

As an educational program, the Professional Doctorate in Engineering (PDEng) in Software Technology (ST) is conducted and assessed by the Eindhoven University of Technology. TU/e dictates certain standards that have to be met. Those standards are mainly related to the design process, project management, and project implementation.

STAKEHOLDER	University Supervisor
ROLE	The university supervisor guards the educational interests of the university and the trainee.
REPRESENTATIVE	Erik de Vink
RESPONSIBILITIES	<ul style="list-style-type: none"> • Monitor, evaluate, assess, and provide regular feedback on the project progress and deliverables • Provide relevant domain knowledge, references, and contacts • Provide relevant information regarding the needs and requirements of the project • Monitor, evaluate, assess and provide feedback on the trainee's design process and qualities of the design • Review the final project report and provide feedback for improvement to fulfill the university standards
ACCEPTANCE (SUCCESS) CRITERIA	<ul style="list-style-type: none"> • Timely report of the project deliverables • Design, implementation, project management, and documentation that meet the PDEng project level
INVOLVEMENT	During the entire project by continuous communication via meetings on ad-hoc basis with the PDEng trainee and regular monthly project steering group meetings.

STAKEHOLDER	PDEng Trainee
ROLE	Software Designer
REPRESENTATIVE	Sodkhuu Dalaikhuu
RESPONSIBILITIES	<ul style="list-style-type: none">• Design, implement, deploy, and test the model-based interface code generator• Apply designer and professional skills• Deliver required results on time
ACCEPTANCE (SUCCESS) CRITERIA	<ul style="list-style-type: none">• Timely report of the project deliverables• Content of sufficient quality as to the level expected of a PDEng trainee
INVOLVEMENT	Full involvement

Appendix B

Appendix 2 Context Diagram

The context diagram, Figure B.1, shows technologies and projects that are relevant to the framework. The Auto Component Interface and the Model-Based Interface Framework are previously developed in-house projects of the Thermo Fisher Scientific company. They provide proof-of-concept for framework design decisions and technology choices. The company built environment includes technologies, tools, and code bases relevant to the current project.

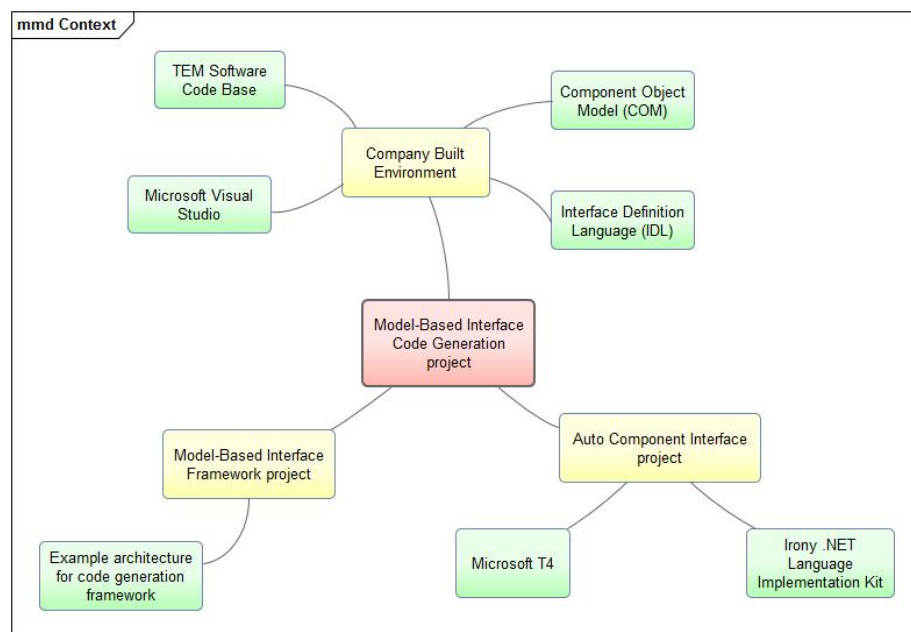


Figure B.1: Context diagram

Appendix C

Appendix 3 Functional Requirements

ID	Description	Priority
FR01	The framework should use a DSL file as input.	MUST
FR01-01	The framework should use IDL as input.	MUST
FR01-02	The framework should use FIDL as input.	OPTIONAL
FR02	The framework should generate code artifacts as output.	MUST
FR02-01	The framework should generate server-side COM-to-C++ wrapper code as output.	MUST
FR02-02	The framework should generate client-side COM-to-C++ wrapper code as output.	OPTIONAL
FR03	The framework should convert COM interface information to object-oriented interface model	MUST
FR03-01	The framework should convert method from source to target	MUST
FR03-02	The framework should convert primitive type parameters of a method from source to target	MUST
FR03-03	The framework should convert classifier type parameters of a method from source to target	SHOULD
FR04	The framework should convert COM enumeration information to object-oriented interface model	SHOULD
FR05	The framework should convert COM class information to object-oriented interface model	SHOULD
FR06	The framework should convert COM dispinterface information to object-oriented interface model	OPTIONAL
FR07	The framework should be able to specify output directory path	MUST
FR08	The framework should be able to generate multiple target code artifact using single source model information	MUST

Appendix D

Appendix 4 Parsed results

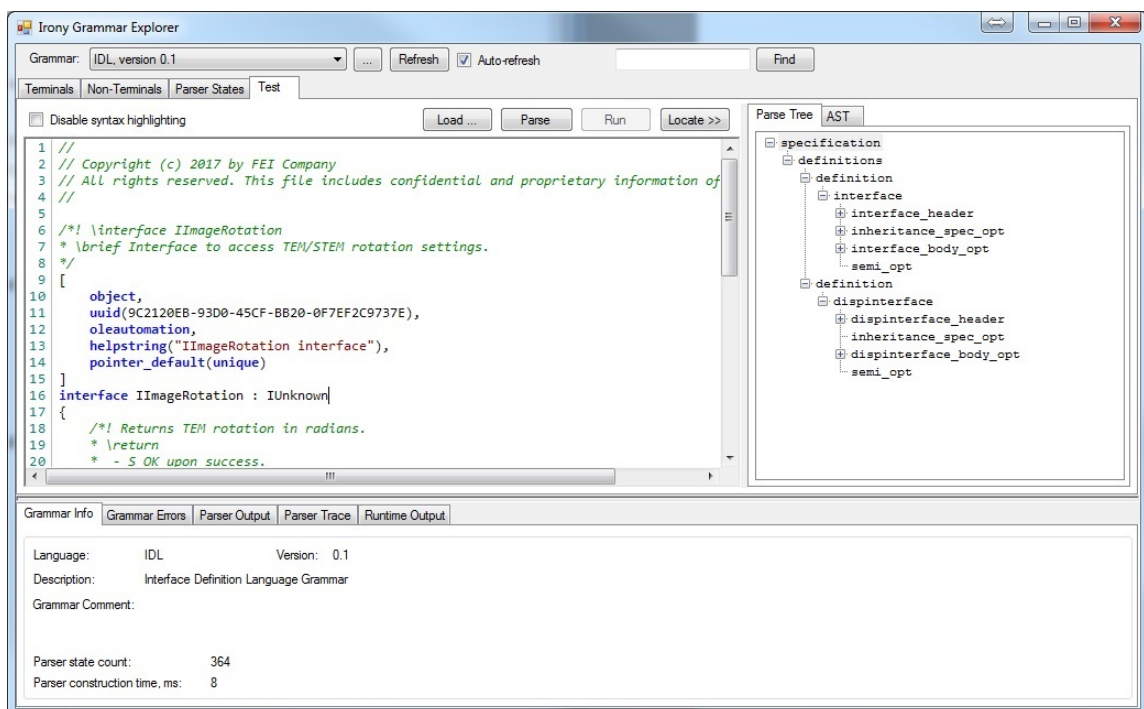


Figure D.1: Parsed result of the IImageRotation.idl file

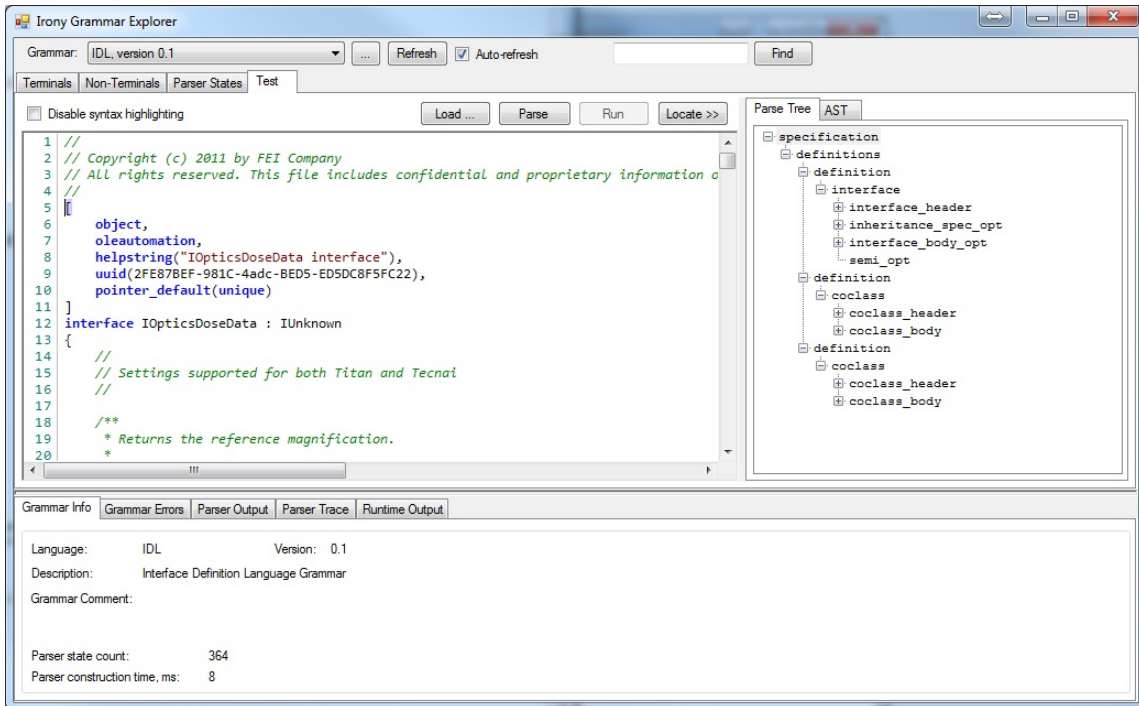


Figure D.2: Parsed result of the IOpticsDoseData.idl file

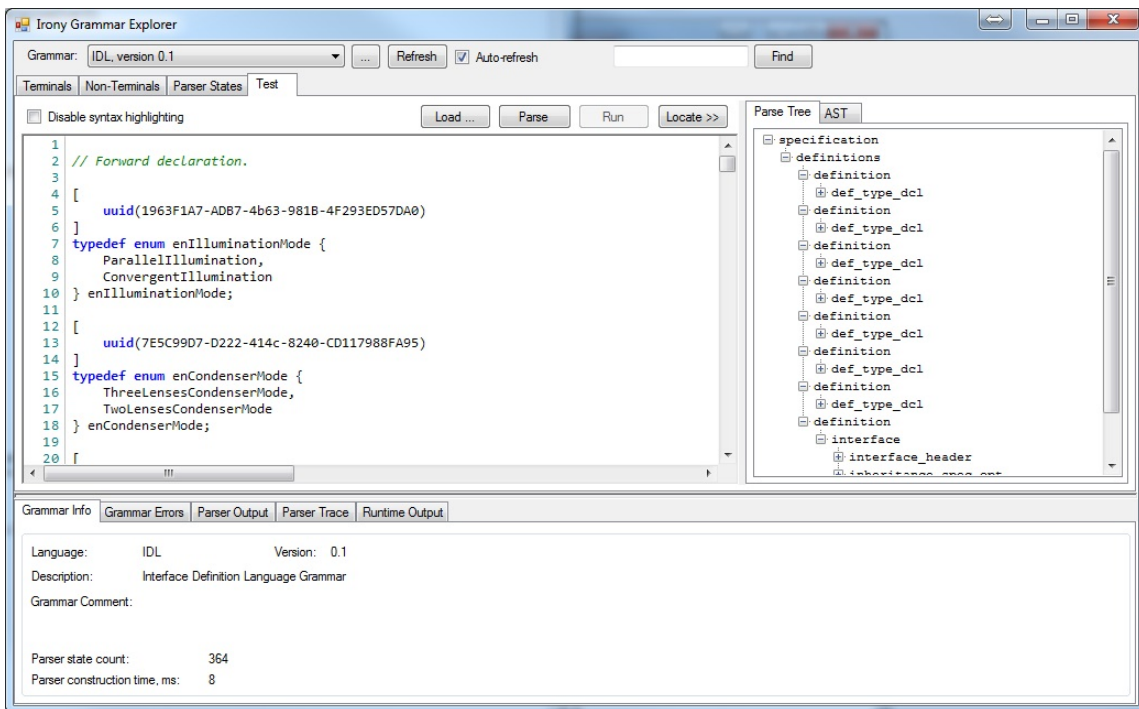


Figure D.3: Parsed result of the ipeo_column.idl file

Appendix E

Appendix 5 Proposal for COM-to-C++ wrapper code generation

E.1 COM header file

```
<#@ include file="common.tt" #>
// This file has been generated using Model-Based Interface Code Generation project.

#pragma once

#include <vector>
#include <Fei/ISupportErrorInfoImplEx.h>
#include <Fei/ComObject.h>
#include "Acquisition/Utilities/Logger.h"
#include "Acquisition/Utilities/DecoupledConnectionPoint.h"
#include "ObjectModel/Com/IomEnergyFilterService.h"
#include "BhvIomAcquisition/<#= itf.Name #>.h"
#include "CoClientOwnedObject.h"

namespace Fei {
    class DispatcherItf;
}
namespace Tem {
}
namespace Iom {

class CoEnergyFilterContext;

class ATL_NO_VTABLE Co<#= itf.Name #> :
    NonCopyable,
    public CoClientOwnedObject<Co<#= itf.Name #>, CComMultiThreadModel>,
    public IDispatchImpl<Com::I<#= itf.Name #>, &__uuidof(Com::I<#= itf.Name #>), &__uuidof
        (Com::__<#= itf.OwnerNamespace.Name #>), /*wMajor =*/ 1, /*wMinor =*/ 0>,
    public ISupportErrorInfoImplEx<Com::I<#= itf.Name #>>,
    public LoggingSupport
{
public:
    static CComPtr<Com::I<#= itf.Name #>> Create(const std::shared_ptr<<#= itf.Name #>>& p<
        #= itf.Name #>, const std::shared_ptr<CoEnergyFilterContext>& pContext);
    virtual void DisconnectObject();

    DECLARE_NO_REGISTRY()

    BEGIN_COM_MAP(Co<#= itf.Name #>)
        COM_INTERFACE_ENTRY(Com::I<#= itf.Name #>)
        COM_INTERFACE_ENTRY(IDispatch)

```

```

    COM_INTERFACE_ENTRY(ISupportErrorInfo)
END_COM_MAP()

DECLARE_PROTECT_FINAL_CONSTRUCT()

<# foreach (var method in itf.Methods)
{
#> SIDMETHOD(<#
    if (method.IsGet()) { #>get_<# }
    else if (method.IsPut()) { #>put_<# }
#><#= method.Name #>(<#
    if (method.Parameters.Count > 0)
    {
        var last = method.Parameters.Last();
        foreach (var parameter in method.Parameters)
        {
            if (parameter.GetParameterType.IsPrimitiveType()) { #><#=
                GetPrimitiveTypeIdentifier(parameter.GetParameterType) #><# }
            else { #>Com::<#= parameter.GetParameterType.Name #><# }
            if (parameter.Pointer) { #>*<# }
            else if (parameter.Pointer2Pointer) { #>**<# }
            #> <#= parameter.Name #><#
            if (!parameter.Equals(last)) {#>,<#}
        }
    } #> );
<# } #>

protected:
    CoEFSlit(const std::shared_ptr<<#= itf.Name #>>& pSlit, const std::shared_ptr<
        CoEnergyFilterContext>& pContext);
    ~CoEFSlit();

private:
    std::shared_ptr<<#= itf.Name #>> m_p<#= itf.Name #>;
    Fei::DispatcherItf& m_dispatcher;

<# foreach (var field in itf.Fields) { #>
    CComPtr<Com::<#= field.GetFieldType.Name #>> m_pI<#= field.Name #>;
<# } #>
};

} // namespace Server
} // namespace Acquisition
} // namespace Fei

```

E.2 COM source file

```

<#@ include file="common.tt" #>
// This file has been generated using Model-Based Interface Code Generation project.

#include "stdafx.h"
#include <Fei/ComApi.h>
#include <Fei/ComObject.h>
#include <Fei/HResultFromException.h>
#include "CoEnergyFilterContext.h"
#include "CoEFSlitEdge.h"
#include "Co<#= itf.Name #>.h"

namespace Fei {
namespace Tem {
namespace Iom {

Co<#= itf.Name #>::Co<#= itf.Name #>(const std::shared_ptr<<#= itf.Name #>>& p<#= itf.
    Name #>, const std::shared_ptr<CoEnergyFilterContext>& pContext) :

```

```

CoClientOwnedObject(pContext->LifecycleSignals()),
LoggingSupport(pContext->Logger()),
m_p<#= itf.Name #>(p<#= itf.Name #>),
m_dispatcher(pContext->BehaviorDispatcher())<#
foreach (var field in itf.Fields) { #>,
m_pl<#= field.Name #>(Co<#= field.GetType().Name #>::Create(m_p<#= itf.Name #>-><#=
    field.Name #>(), pContext))<# } #>
{
    assert(m_dispatcher.IsDispatcherThread());
}

Co<#= itf.Name #>::~~Co<#= itf.Name #>()
{
    assert(m_dispatcher.IsDispatcherThread());
}

void Co<#= itf.Name #>::DisconnectObject()
{
    assert(m_dispatcher.IsDispatcherThread());
}

CCoPtr<Com::I<#= itf.Name #>> Co<#= itf.Name #>::Create(const std::shared_ptr<<#= itf.
    Name #>>& p<#= itf.Name #>, const std::shared_ptr<CoEnergyFilterContext>& pContext)
{
    return CreateComObject<Co<#= itf.Name #>>(p<#= itf.Name #>, pContext).GetInterface<
        Com::I<#= itf.Name #>>();
}

<# foreach (var method in itf.Methods)
{
    #>SIDMETHODIMP Co<#= itf.Name #>::<#
    if (method.IsGet()) { #>get_<# }
    else if (method.IsPut()) { #>put_<# }
    #><#= method.Name
    #><#
    if (method.Parameters.Count > 0)
    {
        var last = method.Parameters.Last();
        foreach (var parameter in method.Parameters)
        {
            if (parameter.GetParameterType.IsPrimitiveType()) { #><#=
                GetPrimitiveTypeIdentifer(parameter.GetParameterType) #><# }
            else { #>Com::<#= parameter.GetParameterType.Name #><# }
            if (parameter.Pointer) { #>*<# }
            else if (parameter.Pointer2Pointer) { #>*<# }
            #> <#= parameter.Name #><#
            if (!parameter.Equals(last)) { #>,<# }
        }
    }
    #>)
try
{
    return ComRetVal(<#
        foreach (var field in itf.Fields) {
            if (field.GetDefinedMethod.Name.Equals(method.Name))
            {
                #>m_pl<#= field.Name #>, <#
                if (method.HasOutRetVal()) {
                    #><#= method.GetOutRetVal().Name #><#
                }
            }
        }
        #>);
}
catch (...)
{
    return LOG_HRESULTFROMEXCEPTION(GetLogger());
}

```



```

}
<# } #>

} // namespace Iom
} // namespace Tem
} // namespace Fei

```

E.3 C++ interface file

```

<#@ include file="common.tt" #>

// Copyright (c) 2009 – 2013 by FEI Company
// All rights reserved. This file includes confidential and proprietary information of
// FEI Company.

#pragma once

namespace Fei {
namespace Tem {
namespace Iom {

<# foreach (var fieldName in itf.Fields.Select(x => x.GetFieldType.Name).Distinct())
    { #>
class <#= fieldName #>;
<# } #>

class I<#= itf.Name #>
{
public:
<# foreach (var method in itf.Methods)
    {
        if (method.HasOutRetVal()) {
            var outRetValType = method.GetOutRetVal().GetParameterType;
            if (outRetValType.IsPrimitiveType()) {
                #> <#= GetPrimitiveTypeIdIdentifier(outRetValType) #><#
            }
            else {
                #> <#= outRetValType.Name #><#
            } #>*<#
        }
        #> <#= method.Name #>() const = 0;
    } #> } #>
};

} // namespace Iom
} // namespace Tem
} // namespace Fei

```

About the Authors



Sodkhuu Dalaikhuu received his Electronic Systems Engineering and Information Technology five-year bachelor's double-degree from the School of Information and Communication Technology of the Mongolian University of Science and Technology (MUST) in June 2014. During his study period, he participated in various domestic and international robotic contests and student research conferences successfully. His final thesis was in speech recognition field and was titled "Research of Mel-Frequency Cepstral Coefficients." Starting from his final year as a student, he has worked as Research Assistant at the MUST and as Embedded Systems Developer at Ametros Solutions Company for one year before becoming a PDEng trainee. His main interest is Embedded Systems and System Architecture.

