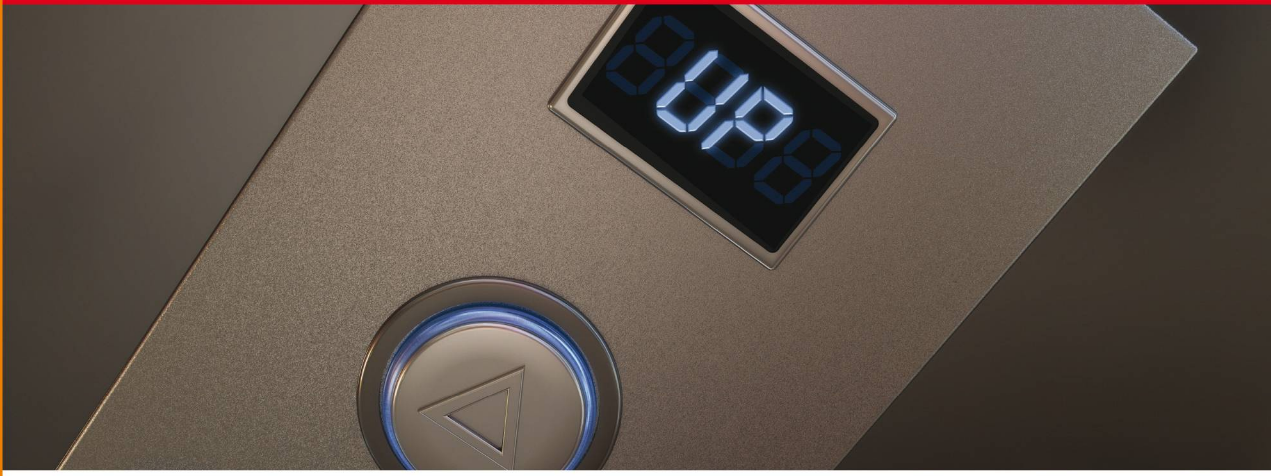
3TU.School for Technological Design

STAN ACKERMANS INSTITUTE

From a data-model to generated access- and store-patterns

Tesfahun Aregawy Tesfay

August 2015



TU Delft
Delft University of Technology

TU/e Technische Universiteit Eindhoven University of Technology

UNIVERSITY OF TWENTE.

From a data-model to generated access- and store-patterns

Eindhoven University of Technology
Stan Ackermans Institute / Software Technology

Partners

ASML Netherlands B.V. Eindhoven University of Technology

Date August 2015

Document Status public

The design described in this report has been carried out in accordance with the TU/e code of scientific conduct.

Contact Address	Eindhoven University of Technology Department of Mathematics and Computer Science MF 7.090, 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 *UniversiteitsDrukkerij*
ISBN	**ISBN: 978-90-444-1382-3**
Abstract	This report describes the design and implementation of a repository generation tool that is used to generate repositories from domain models of the ASML TWINSCAN system. The TWINSCAN system handles a huge volume of data. In the current TWINSCAN SW Architecture, data transfer is combined with control flow. Data transfer to a component that is not under the sender's control must be performed through a common parent in the hierarchy. There are several problems with this approach with respect to execution, encapsulation, and locality of change. These problems drive the need to separate data, control, and algorithms of the scanner's software architecture. To tackle these problems, the main objective of this project was to design and implement a repository generation tool for generating data repositories from domain models. The structure of this data is defined by a domain model in an implementation independent formalism. The tool supports several flavors of repositories. As a result of the flexibility of the architecture, it is possible to switch between technologies and implementation patterns without touching domain models. The repository generation tool is tested through continues architecture and design reviews by supervisors, unit tests, and tests by stakeholders in the real environment. The results obtained in this project are being used in an active ASML project within the Metrology group. The results have improved productivity and increased efficiency.
Keywords	Model-driven architecture, model-driven engineering, PDEng, domain models, implementation models
Preferred reference	T.A. Tesfay **From a data-model to generated access- and store-patterns**, SAI Technical Report, August 2015. (**ISBN: 978-90-444-1382-3**)
Partnership	This project was supported by Eindhoven University of Technology and ASML.
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 ASML. The views and opinions of authors expressed herein do not necessarily state or reflect those of the Eindhoven University of Technology or ASML, and shall not be used for advertising or product endorsement purposes.
Disclaimer Liability	While every effort will be made to ensure that the 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 service marks of their respective owners. We use these names without any particular endorsement or with the intent to infringe the copyright of the respective owners.
Copyright	Copyright © 2015. 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 ASML.

Foreword

ASML has become a large company in many aspects such as the number of systems being sold, the amount of complexity handled within the system's design and the number of employees working on it. It is a well-known fact that growth comes with the challenge of remaining agile. In order to remain competitive, an efficient design and production process is of outmost importance. With respect to software, this means that the effort, needed to get from a conceptual idea to an implementation installed on a system, should be as small as possible.

Within the software architecture group, a number of architects have been investigating the application of Model Driven Engineering methods and tools to improve the efficiency of creating software. One of the areas being investigated is the domain of data modeling as executing ASML systems create and manipulate a lot of data. In order to support the design of data models, a prototype has been developed within ASML that allows definition of data structures with their relations and generation of repositories that can be installed on the system.

In order to become usable for a large population of software designers, the prototype needs to be matured into a production worthy tool. As the SW architects have been very busy keeping the prototype running and supporting its users, insufficient resources were available to mature it. That is the moment we decided to define an OOTI assignment which attracted the attention of Tesfahun.

The assignment consisted of reevaluating the requirements imposed on such tool, creating a good design and implementation of a tool that allows definition of data structures and generation of an implementation of the data structures and the repositories. Not a small task.

Tesfahun decided to take the challenge and apply for the task. He got invited for an application interview where the assignment was explained. Besides discussing the assignment, also an impression of the candidate needs to be gained. Tesfahun plays soccer and when I asked about it he told me that he plays as defender. When confronting an attacker, he literally stated: "… either the man has to go or the ball has to go..." That was the moment that we were convinced about his motivation and decided to accept his application.

During the assignment he has clearly communicated how he would be able to contribute. That led to adjusting the assignment such that more focus has been put on the code generation part. One project within ASML was selected to utilize the prototype for data design. Though in general it is not advised to depend on the work of a student for a product that needs to be delivered in time to customers, we decided to use the work of Tesfahun within the context of an actual ASML project.

The ASML team executing the project has put real pressure on Tesfahun and his flexibility and motivation have been stress tested as well as his product. The team is really satisfied and they are appreciating the benefits of the improved productivity. Large quantities of code are being generated now that otherwise would need laborious manual typing.

Given his motivation and ambition, we are sure that Tesfahun is a valuable asset for every project and we are confident that this young man will mature into a very skilled software designer and capable architect.

Wilbert Alberts and Ronald Koster
SW architects ASML.

Preface

This report describes the results of the project ‘From a data-model to generated access- and store-patterns’ carried out by Tesfahun Tesfay at ASML, Veldhoven, The Netherlands. This project is performed over the last nine months as a partial fulfillment to obtain the Professional Doctorate in Engineering (PDEng) degree in Software Technology, from the Eindhoven University of Technology.

The main objective of this project is to design and implement a repository generation tool for generating repositories from domain models based on configuration settings. This repository generation tool is built based on a flexible architecture. This flexible architecture allowed separation of domain models from repository implementation technology details. Switching between different implementation patterns and technology choices is easily possible without touching domain models.

This project is carried out by the author, Tesfahun Tesfay, under the supervision of Ronald Koster and Wilbert Albers from ASML and Tim Willemse from the Eindhoven University of Technology.

The report is intended for everyone who is interested in the application of model-driven architecture to tackle data aspects of complex systems. Basic understanding of software engineering, model-driven architecture, and model-driven engineering is assumed.

This report is organized such that readers are guided smoothly from the problem through to the solution. The first four Chapters present the context, the thorough analysis of the problem, the domain, and the requirements consecutively. In Chapter 5, the architecture of the repository generation tool is described. In this chapter, the goal of the individual components and the relationships between them is described. In Chapter 6, each of these components in the architecture is opened up and discussed in detail. In both chapters, the architectural and design tradeoffs are documented. In Chapter 7, relevant notes are given about the implementation of each individual component in the architecture. Chapter 8 presents the testing techniques applied to ensure the quality of the tool and support its evolution in the future.

In Chapter 9, the results obtained in this project are summarized. The features that should be supported in the future are also presented. In Chapters 10, the project management strategy applied in this project is discussed. In Chapter11, technical reflections on the design criteria that are selected in this project are presented. A brief person reflection of the author on the organizational and technical aspects of the project is also presented.

Tesfahun Aregawy Tesfay
Date August, 2015

Acknowledgements

With this project, I conclude the two years PDEng degree program, commonly known as OOTI, at the Eindhoven University of Technology (TU/e). This program has given me the opportunity to tackle several real world problems by applying the knowledge I have gained from education through the years. Excluding workshops and assignments, the project described in this report is the fifth project I have worked on over the course of this program. I would like to thank the people who contributed to the successful completion of the program as well as this project.

I would like to thank my supervisors at ASML: Wilbert Alberts and Ronald Koster. Without your continuous guidance and support, the successful completion of this project would not have been possible. The feedback and discussions during the weekly and PSG meetings were invaluable.

I would like to thank my TU/e supervisor, Tim Willemse, for the motivation and useful feedback he has given me over the course of this project. Thank you for taking time out of your busy schedule for coming to ASML during every PSG meeting.

I would like to thank Matija Lukic and Sander Kersten for contributing their knowledge regarding the generated repositories, and for using, and testing my tool starting from day one. It was a great feeling to see you happy and using my tool in a real ASML project. Thank you and all of your colleagues for having me in your motivating workplace during my collocation.

I would like to thank Sven, Sofia, Ashu, Theo, Niels, Yuri, and others who contributed their domain knowledge and experience. I would also like to thank Rogier Wester for allowing me to carry out this project in his group.

I would like to express my deepest gratitude to the PDEng program director, Ad Aerts, for giving me the chance to face the challenging roles and projects I am interested in. I would like to thank all the PDEng coaching staff for helping me in different professional skills along the way. I would like to thank Maggy de Wert for dedicating her time to answer my questions regarding all kinds of procedures and for the motivation she has given me during my stay at the TU/e. I would also like to thank my friends and colleagues from the OOTI.

I would like to thank my mother, Mebrhit, and all my family for their support and prayers. I would like to thank my wife, Maereg (Magi), for her patience and support every single day. I would like to thank my beautiful daughter, Bethany (Betu), for joining my family and for being such a nice daughter. I am truly blessed to have you in my life.

I thank God for giving me wisdom and strength to successfully complete the program.

Tesfahun Aregawy Tesfay
Date August, 2015

Executive Summary

ASML is the leading provider of lithography systems in the world. These lithography systems are complex machines that are critical to the production of integrated circuits (ICs) or chips. The TWINSCAN system is the most important product line of the ASML lithography systems. The ASML TWINSCAN produces up to 200 wafers per hour. These wafers are 300 mm diameter and are exposed at 22 nm resolution.

The TWINSCAN system handles a huge volume of data. In the current TWINSCAN SW Architecture, data transfer is combined with control flow. Data transfer to a component that is not under the sender’s control must be performed through a common parent in the hierarchy. There are several problems with this approach with respect to execution, encapsulation, and locality of change. These problems drive the need to separate data, control, and algorithms of the scanner’s software architecture.

To tackle the data handling problems, the main objective of this project was to design and implement a repository generation tool for generating data repositories from domain models. The tool is accompanied by a means to flexibly select from a set of implementation patterns, allowing the generation of an implementation of data repositories and access interfaces from a domain model. The structure of this data is defined by a domain model in an implementation independent formalism. As a result of the flexibility of the architecture, it is possible to switch between technologies and implementation patterns without touching domain models. This tooling support reduces development time and increases efficiency.

The repository generation tool consists of an Implementation Model Language that helps users specify choices of implementation patterns without polluting their domain models with implementation details. To maximize flexibility, this language is based on the recipe-ingredient relationship in a traditional cookbook. To maximize productivity and facilitate learning the Implementation Model Language and its syntax, the tool contains an Implementation Model Wizard capable of creating initial implementation models from domain models. To early discover errors in the implementation model before code generation, the tool is equipped with an Implementation Model Validation. This protects the tool from producing a code that does not compile or a wrong code that compiles. The tool consists of a repository generator component to allow generation of repositories from domain models based on the recipes in implementation models. This is realized by providing several code generation modules.

The repository generation tool is tested through continues architecture and design reviews by supervisors, unit tests, and tests by stakeholders in the real environment.

The results are being used in an active ASML project within the Metrology group. The productivity of the group is significantly improved. They have already generated 600+ files of C++ code using the tool. Manipulation of domain models is very easy with the repository generation tool. The effort and time required to see changes in a domain model reflected in the generated code is reduced to a one button click.

Table of Contents

List of Figures

List of Tables

1.Introduction

This chapter provides the context for this project with a brief introduction to ASML, their most important product, and the software architecture of this product. This chapter also gives the outline of this report.

1.1 Context

ASML is the leading provider of lithography systems in the world [1]. These lithography systems are complex machines that are critical to the production of integrated circuits (ICs) or chips. The TWINSCAN system is the most important product line of the ASML lithography systems.

Manufacturing ICs in the semiconductor industry requires a number of process steps, from slicing a cylinder of purified silicon material into wafers through to packaging, as shown in Figure 1. A wafer is a sliced and polished silicon material on which layers of images of patterns are created during exposure. These images of patterns are contained in a flat quartz plate called a reticle. The ASML TWINSCAN system performs one of the process steps of the IC manufacturing process, called the lithography process, i.e. Step 5. The TWINSCAN system is responsible for exposing wafers as quickly and as accurately as possible based on the performance specifications: productivity, overlay, and imaging (resolution).

The ASML TWINSCAN produces up to 200 wafers per hour. These wafers are 300 mm diameter and are exposed at 22 nm resolution.

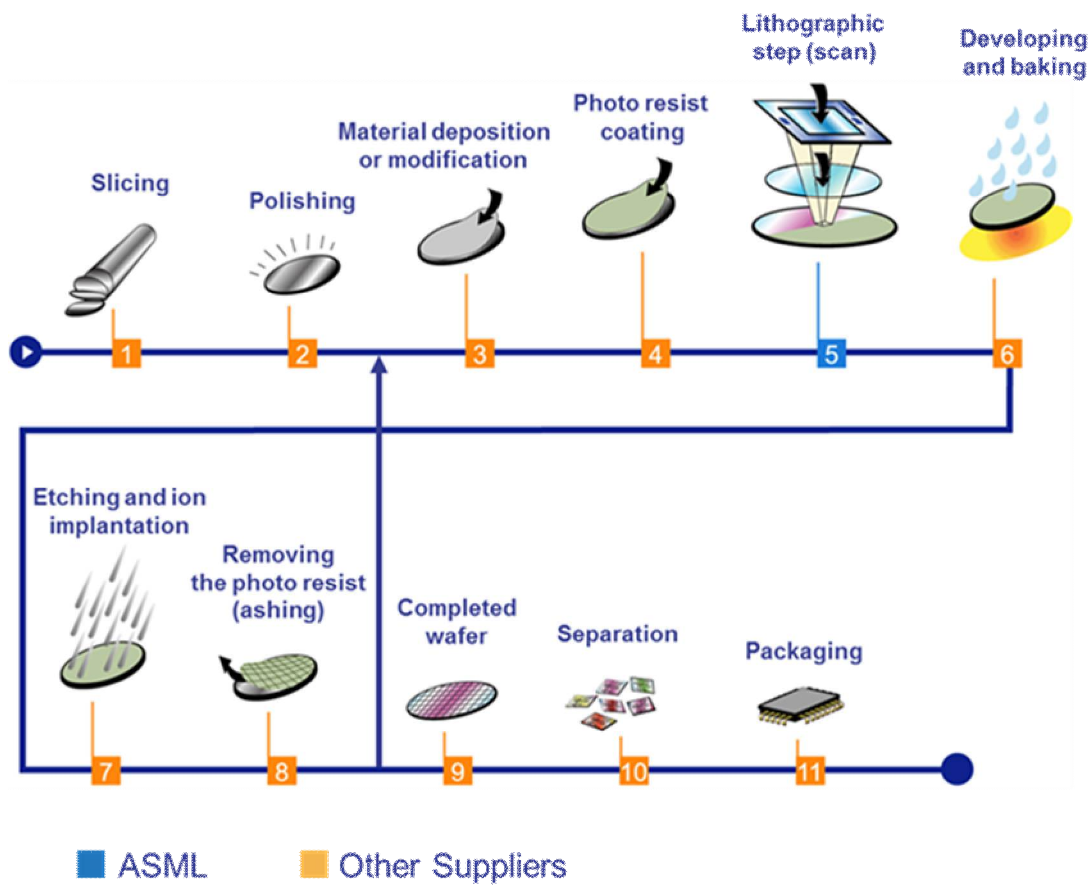


Figure 1: IC Manufacturing process, *showing the life of a wafer.*

The TWINSCAN machine contains two stages that are used for positioning wafers. A stage can be either at the measurement (metrology) station or at the exposure station at a time, as shown in Figure 2. At the measurement station, a wafer is measured in XY and Z directions. Metrology is the science of this measurement. The result of this measurement is used at the exposure station to expose a layer on a wafer correctly based on an image of a pattern on a reticle. Each layer can be repeated for a group of wafers. This group of wafers is called a lot.

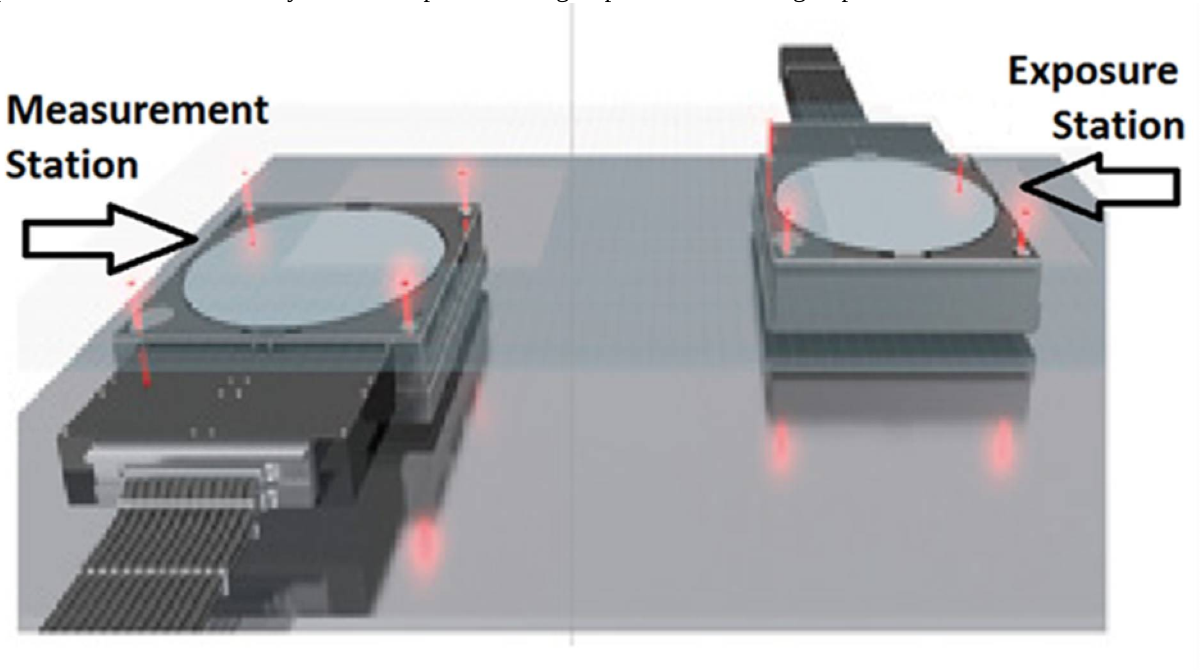


Figure 2: The TWINSCAN dual stage system, *one wafer in the measure station and one wafer in the expose station.*

1.2 The TWINSCAN SW Architecture

The TWINSCAN Software Architecture supports the operations of the TWINSCAN system. These operations include wafer measurement and exposure, calibration, diagnostics, and scheduling of tasks within the TWINSCAN system. Furthermore, it provides interfaces to the external environment.

The TWINSCAN software architecture is organized into Functional Clusters (FC), Building Blocks (BB), Components (CC), Layers (LA), Release Parts (RP), and Assemblies (AS). The component (CC) is the most relevant part of this organizational structure.

1.2.1. Components (CC)

An ASML software component (CC) is a basic unit of the TWINSCAN software development. A CC may correspond to a physical structure of the TWINSCAN or to general purpose functionality. CC is contained in exactly one Layer (LA) and exactly one Building Block (BB). Software components are assigned to layers based on their responsibility. For example, the software components responsible for controlling the flow of tasks are assigned to the Controllers Layer. The components responsible for measurement and exposure of wafers are assigned to the Metrology Layer.

1.3 Problem Area

The problem that we tackled in this project involves data aspects of the TWINSCAN software architecture. The goal of this project is to improve data flow, storage, and sharing within and across ASML software components in the TWINSCAN software architecture. This problem is discussed in detail in Chapter 2.

1.4 Outline

This report is further structured as follows:

Chapter 2 provides the problem analysis, the project stakeholders, and an overview of the design opportunities and challenges in this project.
Chapter 3 presents the result of a thorough analysis of the problem domain.
Chapter 4 presents the requirements for this project.
Chapter 5 describes the high level system architecture and architectural tradeoffs made in this project.
Chapter 6 discusses the detailed design of the components in the reference architecture. The tradeoff design decisions that guided the design process are also documented.
Chapter 7 explains the implementation aspects of the system.
Chapter 8 explains the testing strategies applied in this work.
Chapter 9 concludes this work.
Chapter 10 presents the project management, planning, and risk management strategies applied in this work.
Chapter 11 reflects on the development process of this project. The design criteria selected for this project are also revisited.

2.Problem Analysis

The problem area that we tackled in this project is introduced in Chapter 1. The purpose of this chapter is to discuss data handling in the TWINSCAN SW Architecture, the problems associated with data handling, the proposed solution direction that initiated this project, as well as to present the main objectives of this project. The stakeholders and their intentions are discussed. The identified early design opportunities and challenges are also presented.

2.1 Data Handling in the TWINSCAN SW Architecture

In the current TWINSCAN SW Architecture, data transfer is combined with control flow. Furthermore, data transfer to a component that is not under the sender's control must be performed through a common parent in the hierarchy. A simplified version of a common example of this situation is shown in Figure 3. Input data, required to process a lot, is given through the main controller component. This input data is pushed down to the sub-controller component. Part of this input data is required in the measurement station, and the other part in the exposure station. The sub-controller component pushes the right input data down to the right components.

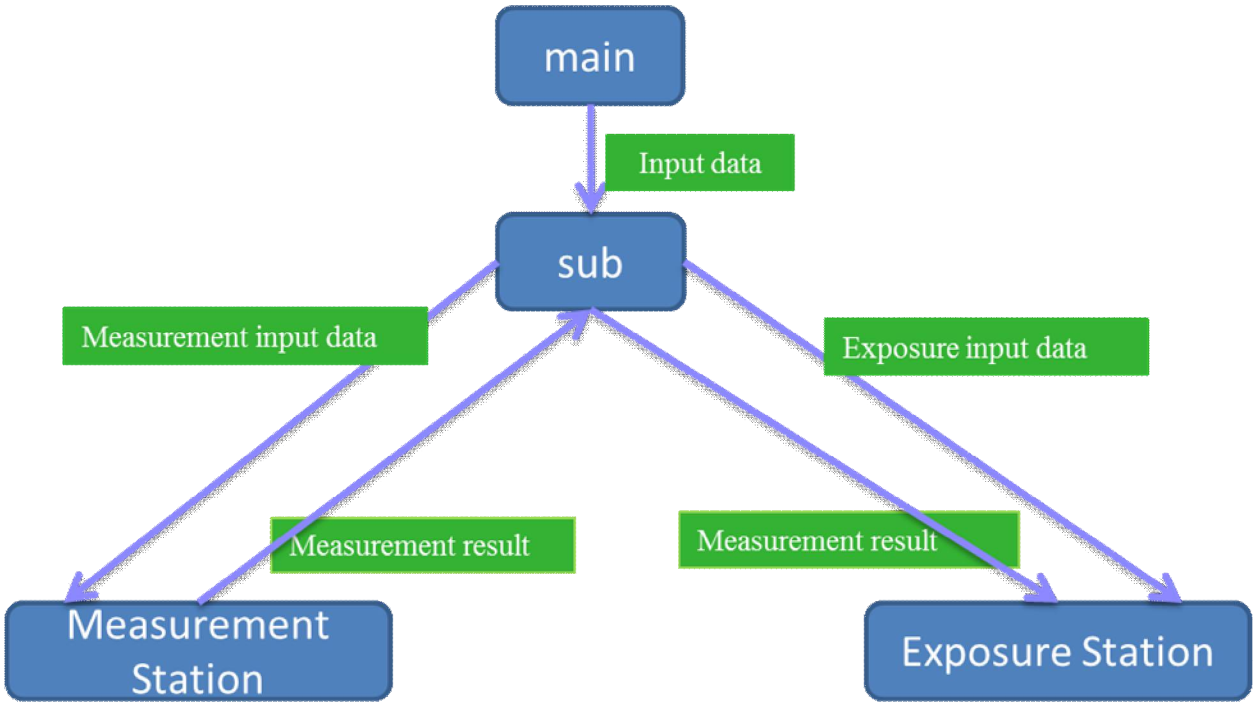


Figure 3: Data Handling in the TWINSCAN SW Architecture, *the purple arrows represent data and control flows combined.*

The software components in the measurement station, measure wafers and store the measurement results for a later use. This measurement result is required by the components in the exposure station to accurately expose the wafer. Therefore, this measurement result needs to be transferred from the measurement station to the exposure station. According to the current TWINSCAN SW Architecture, the sub-controller component pulls the measurement result up from the measurement station and pushes it down to the components in the exposure station. This is because the sub-controller component is the common parent of the measure station and exposure station. There are several problems with this approach. The main ones are described below.

- **Execution**

Data is copied many times from the producer component to the consumer component. This creates a direct impact on the CPU load, memory, disk, and network resources.

- **Encapsulation**

Since the information needed at lower level components is also known at higher level components, it is not easy to verify whether high-level components are not using a data intended for lower level components.

- **Locality of Changes**

A data related software change in one component propagates to many components. For example, changing the measurement result data structure in the components of the measurement station causes the sub-controller component to change.

In the TWINSCAN system, since there are several components involved in the data transfer, these problems are far worse than what is depicted in Figure 2.

2.2 Separation of Data, Control, and Algorithms

To tackle the problems associated with the data handling in the current TWINSCAN SW Architecture, the solution direction shown in Figure 4 is proposed. The proposed solution is based on the separation of control services, durative services, and domain services.

- **Control Services**

Control services determine the execution order of tasks in the system. Control services are designed by using state machines. Control services instruct durative services, and request the creation and destruction of data objects from domain services. Control services request decision values from domain services. Control services are also responsible for the availability of data required by durative services.

- **Durative Services**

Durative services are algorithms and hardware actions that take time and tasks that need and produce data. Durative services store and retrieve data by using domain services.

- **Domain Services**

Domain services implement data storage, retrieval, concurrency, persistence, integrity, and transactionality based on the domain models of the TWINSCAN system.

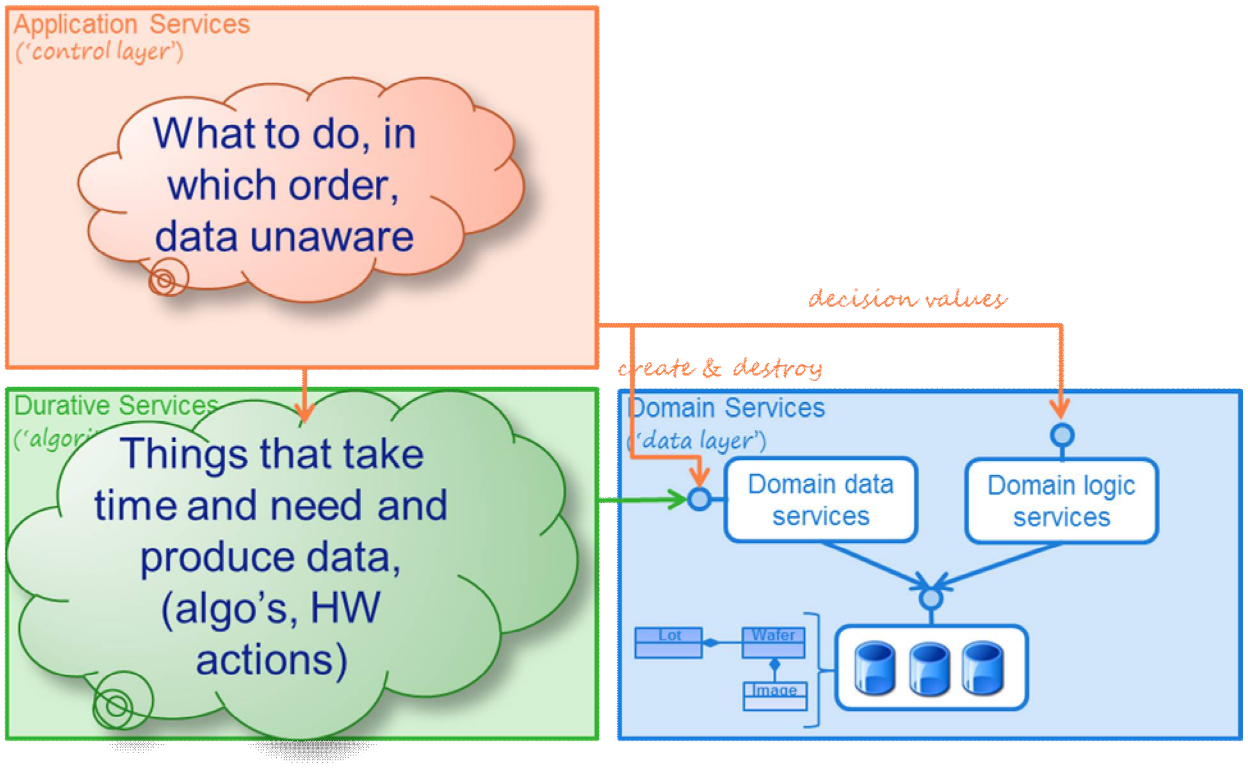


Figure 4: The Proposed Solution Direction *based on separate control, durative, and domain services.*

The main goal of the separation of data, control, and algorithms is to tackle the problems associated with data handling, such as execution, encapsulation, and locality of change. In this approach, the measure input data, exposure input data, and measurement results are stored in their respective repositories, as shown in Figure 5. The sub-controller component is not bothered by data transfer anymore except for IDs. The main controller component directly stores the measurement input data in the measurement data repository and the exposure input data in the exposure data repository. Upon receiving the IDs, the components in the measurement and exposure stations access the required input data and measurement result from their respective repositories. While synchronization and life cycle management may be an issue in the new design, we believe that the benefits of the separation of data and control outweigh the constraints introduced.

Unlike the previous approach, data is not being copied unnecessarily from component to component. We see that the measurement input data, exposure input data, and measurement results are not being copied to the sub-controller component. This improves the execution cost regarding CPU load, memory, disk, and network resources. Data encapsulation has also improved. This is because components only access data that is intended for them. Furthermore, changes to the measurement result data structure in the measurement station do not cause the sub-controller component to change. This way localization of change is improved.

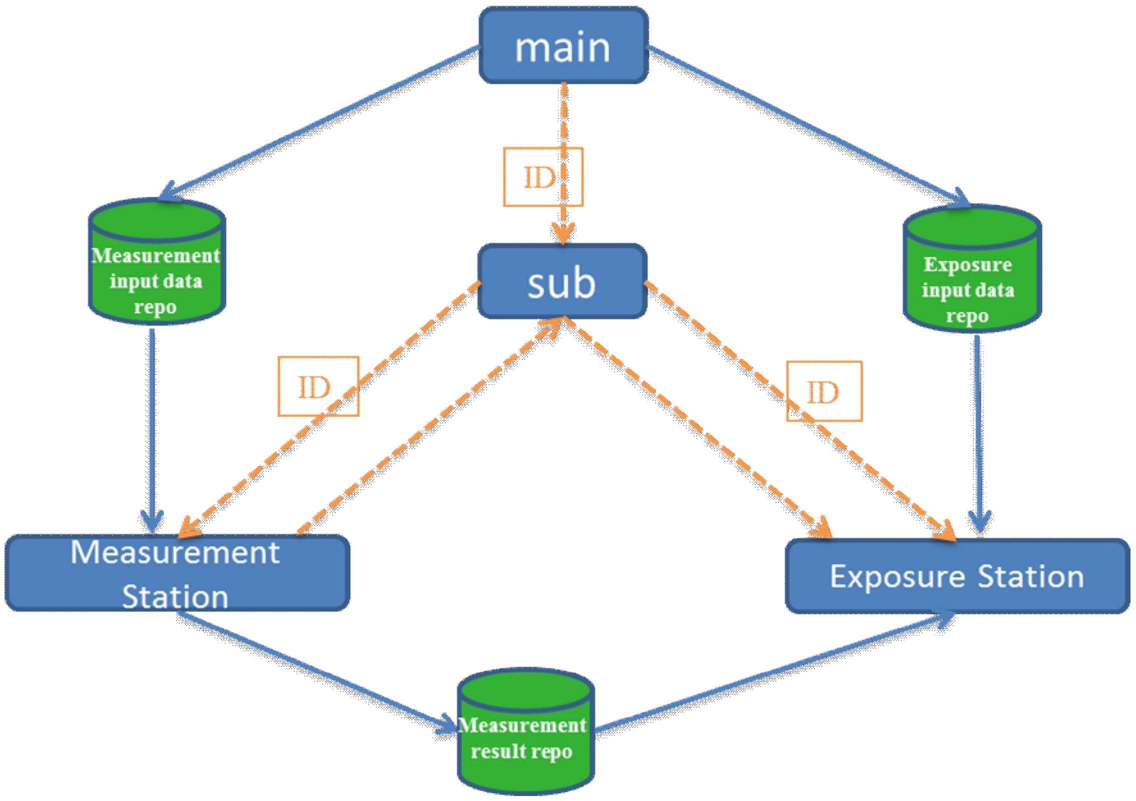


Figure 5: Data handling, *blue arrows represent data flow and orange arrows represent control flow with and/or without IDs*

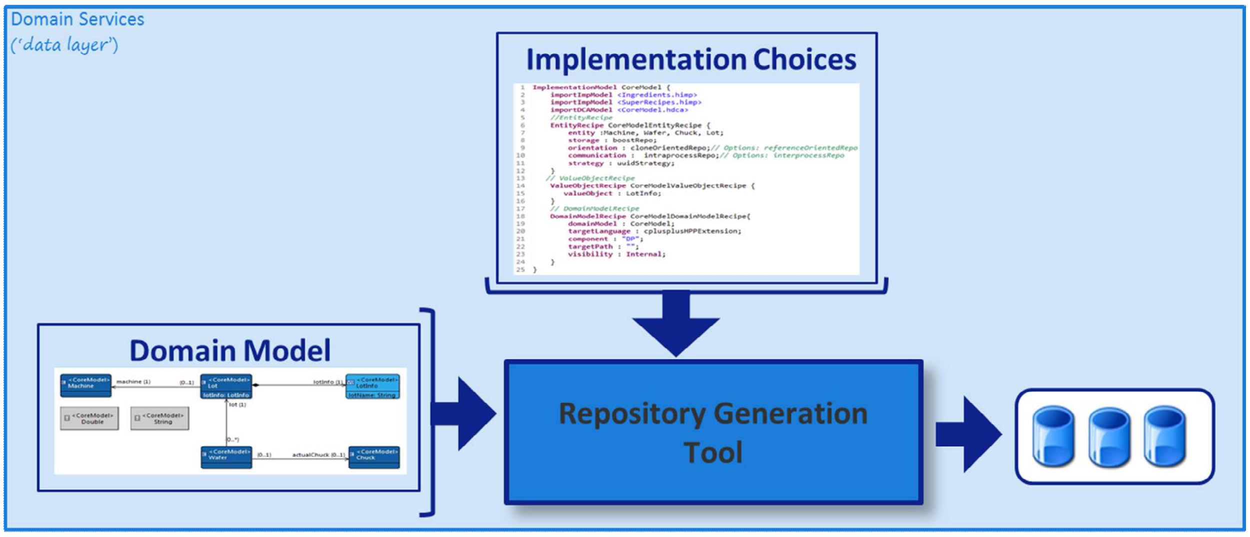
The separation of data, control, and algorithms has initiated this project. The scope of this project is within the domain services. In this approach, domain data models, domain models for short, need to be defined. To realize sharing data between and/or within processes at runtime, instances of the domain model need to be stored in repositories.

2.3 Project Objective

The TWINSCAN system handles a huge volume of data. To tackle the problems associated with the current data handling architecture of the TWINSCAN system, data is stored in repositories. The structure of this data is defined by a domain model. A working TWINSCAN contains the repositories holding the data as defined by this domain model.

This approach is supported by a tool that allows designers to define domain models in an implementation independent formalism and generate the implementation. This tooling support reduces development time and increases efficiency.

The main objective of this project is to design and implement a repository generation tool for generating repositories from domain models based on implementation choices made by users, as depicted in Figure 6. This repository generation tool must be based on a flexible architecture. This flexible architecture keeps domain models separated from repository implementation technology details. Switching between different repository implementation flavors and technology choices must be easily possible without touching domain models.



```
ImplementationModel CoreModel {
    importImpModel <Ingredients.himp>
    importImpModel <SuperRecipes.himp>
    importDCAModel <CoreModel.hdca>
    //EntityRecipe
    EntityRecipe CoreModelEntityRecipe {
        entity :Machine, Wafer, Chuck, Lot;
        storage : boostRepo;
        orientation : cloneOrientedRepo;// Options: referenceOrientedRepo
        communication :  intraprocessRepo;// Options: interprocessRepo
        strategy : uuidStrategy;
    }
   // ValueObjectRecipe
    ValueObjectRecipe CoreModelValueObjectRecipe {
       valueObject : LotInfo;
    }
    // DomainModelRecipe
    DomainModelRecipe CoreModelDomainModelRecipe{
        domainModel : CoreModel;
        targetLanguage : cplusplusHPPExtension;
        component : "DP";
        targetPath : "";
        visibility : Internal;
    }
}
```

Figure 6: Project Objective

2.4 Stakeholders

The stakeholders involved in this project are shown in Figure 9. The interests of these stakeholders and their representatives are also discussed.

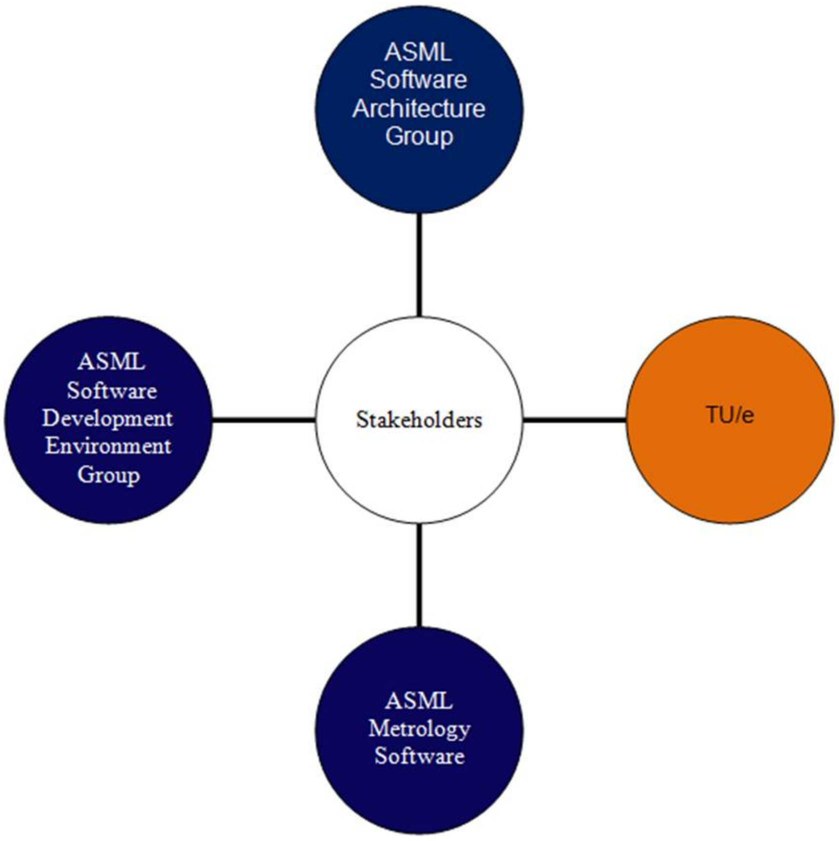

ASML Software Architecture Group
ASML Software Development Environment Group
Stakeholders
TU/e
ASML Metrology Software

Figure 7: Stakeholders

2.4.1. ASML Software Architecture Group

This project is carried out at ASML within the Software Architecture Group. The ASML Software Architecture Group is responsible for defining, maintaining, and improving the TWINSCAN software architecture, and introducing new efficient technologies. In this project, they are interested in improving the efficiency of data handling in the TWINSCAN system. They are represented by the supervisors from ASML (Ronald Koster and Wilbert Alberts) and Sven Weber. As a data architect, Ronald Koster is one of the main users of the repository generation tool. For example, he uses the tool to enforce architectural rules.

2.4.2. TU/e

TU/e is the main stakeholder of the execution process of this project. They are interested in the technological design of the project, the criteria used to evolve the design, and the final report. The interests of the TU/e are represented by the university supervisor, Tim Willemse, the trainee, Tesfahun Tesfay, and the program director, Ad Aerts.

2.4.3. ASML Metrology Group

The ASML Metrology Group is responsible for the measurement and correction of the position of wafers for accurate exposure. Software architects and software engineers within the Metrology department are the main users of this tool. In this project they are represented by Matija Lukic, Sofia Szpigiel, and Sander Kersten. They are the main stakeholders for the generated code.

2.4.4. ASML SW Development Environment Group

The ASML Software Development Environment Group is responsible for the deployment and integration of the TWINSCAN software tooling. In this project, they are interested in the ability of the tool to be deployed in the Eclipse-based WindRiver Workbench. They are represented by Sander Van Hoesel and Ruud Goossens.

2.5 Design Opportunities

The most important design opportunities and challenges identified in this project are: flexibility, reusability, and scalability. These design challenges were identified through analysis of requirements, problem domain, and discussions with stakeholders. During the identification and selection of these design challenges, the criteria for the evaluation of technological designs described in [2] are also considered. These criteria are used throughout the course of the project to improve the design of the repository generation tool.

- **Flexibility**

To reduce the complexity of software change, flexibility is required with respect to how data is handled in the TWINSCAN software architecture. Keeping proper coupling between domain and technology concepts is one of the most important design challenges in this project. It is necessary that domain models are decoupled from repository implementation and technology details. The repository generation tool must allow users to flexibly select from different repository implementation and technology choices.

- **Reusability**

The design challenge here is to realize code generation with a minimum effort. This is achieved by reusing existing model fragments as much as possible.

- **Scalability**

The scalability design challenge is identified to allow the solution to handle bigger domain models of the TWINSCAN system regarding data.

Economical Realizability and Societal Impact [2] are selected as non-relevant for this project. Since the project was based on a fixed budget, the analysis of financial implications was not necessary. Therefore, economical realizability was not relevant for this project. The health hazard prevention mechanisms of the TWINSCAN system are implemented in the hardware. Since this is a software only project, societal health and well-being analysis was not necessary. Therefore, Societal Impact was selected as the second non-relevant criterion for this project.

3.Domain Analysis

The problem analysis is described in Chapter 2. The purpose of this chapter is to present the result of a thorough analysis of the domain.

3.1 Domain Model

Domain models are at the heart of the domain services. A domain model captures the relevant data concepts of the lithography process executed on the TWINSCAN machine. The model also captures the relationships between these data concepts. The ideas behind these modeling concepts and relationships are inspired by the principles of domain driven design [3].

3.2 Domain Model Language

To simplify modeling data aspects of the TWINSCAN system, ASML is developing a domain model language specifically for modeling data. The development of this language is outside of the scope of this project. However, the thorough identification of the requirements for this language has been within the scope of this project. These requirements are shown in Appendix 5. The development of this language has continued to mature throughout the course of this project. Graphical and textual syntaxes are defined for this language. The core domain model in Figure 11 and non-core domain model in Figure 12 are modeled by using this domain model language.

The Metamodel of domain model language, showing the total structural class hierarchies is shown in Figure 8.

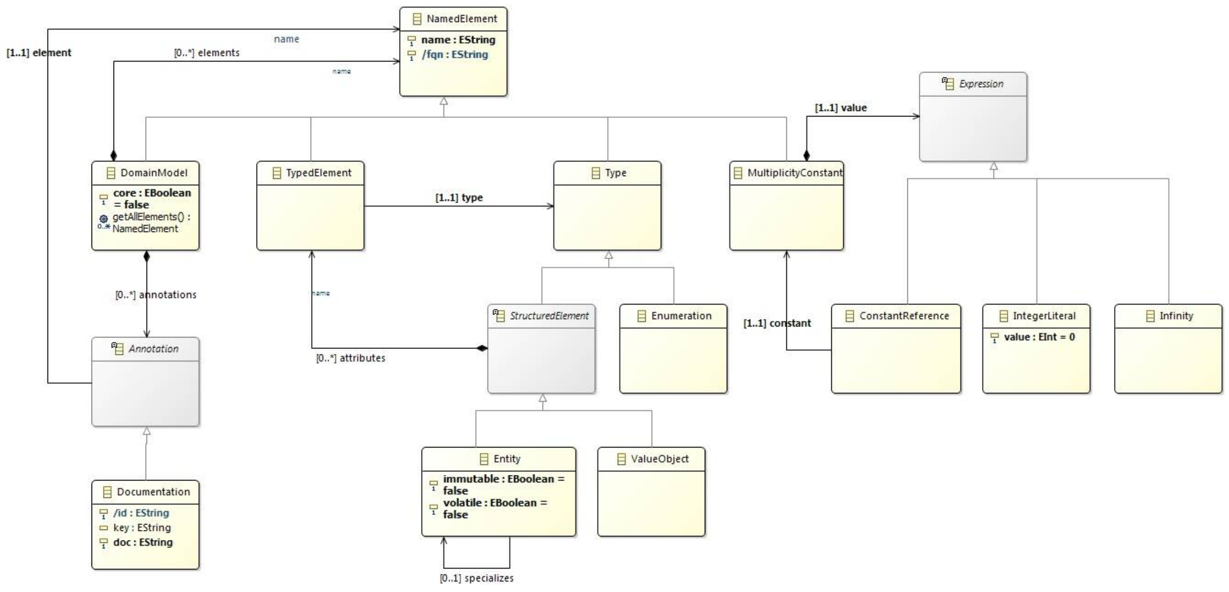


Figure 8: Domain Model Language Metamodel, *class hierarchy.*

The Metamodel of the attributes and associations of this language is shown in Figure 9. Associations and attributes are TypedElements. These associations and attributes have Multiplicities.

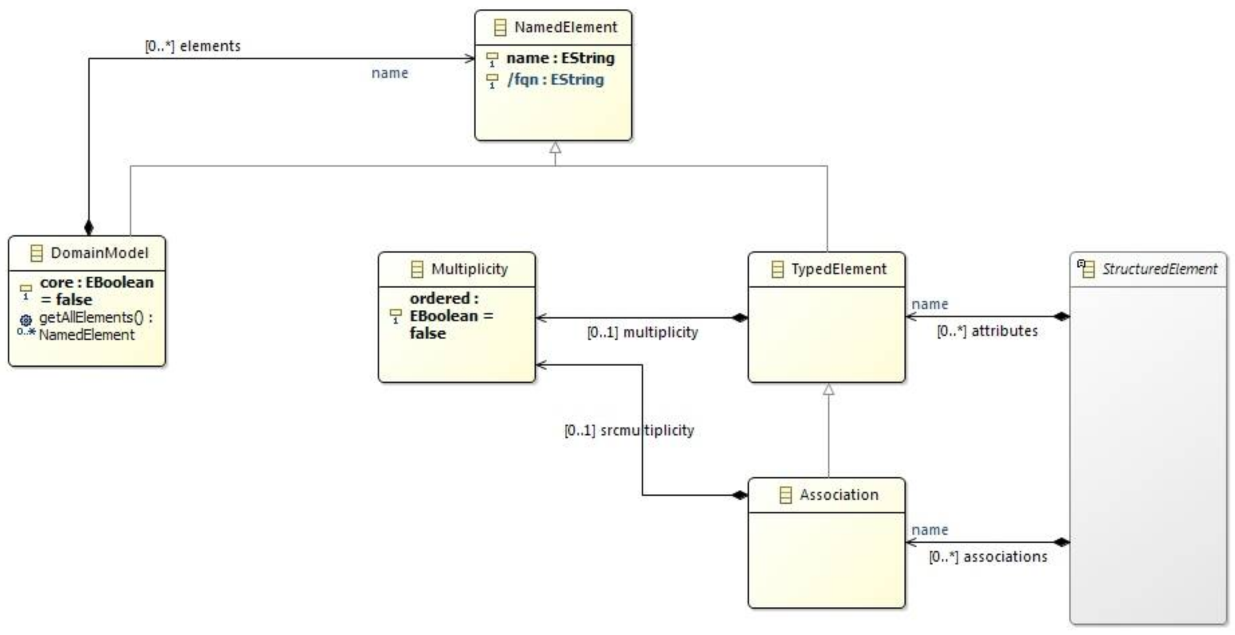


Figure 9: Domain Model Language Metamodel, attributes and associations class hierarchy

The Metamodel of the multiplicities used to model the association ends between different data concepts is shown Figure 10.

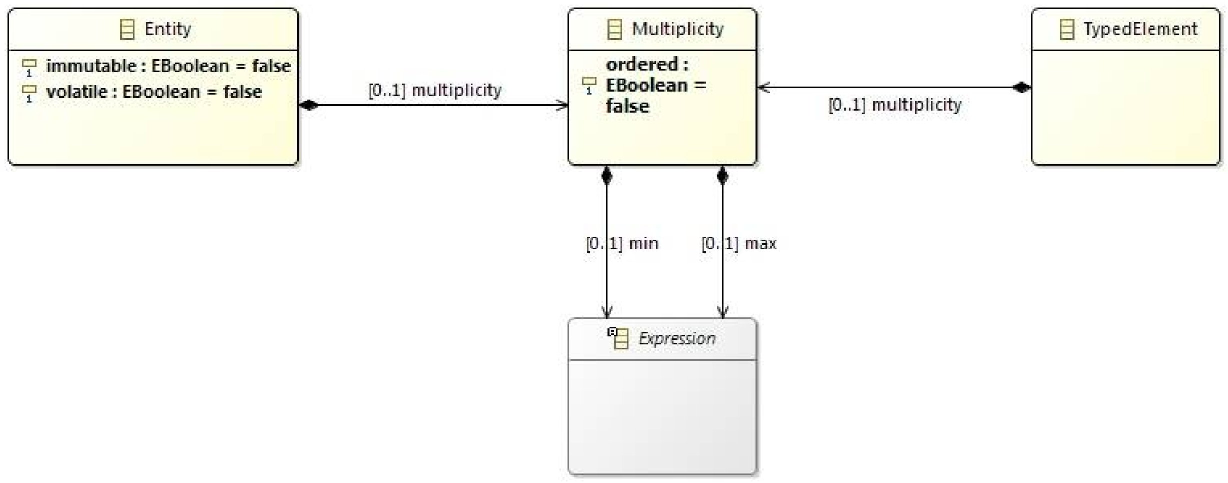


Figure 10: Domain Model Language Metamodel, multiplicity class hierarchy

Since the inputs for the repository generation tool are domain models written in the domain model language, the relevant domain model and language concepts are explained in the upcoming sections.

3.3 Domain Model Concepts

Domain models are composed of a number of data concepts of the TWINSCAN system. These data concepts are conceptually different and must be handled differently. For example, Lot and Lotinfo in the domain model in Figure 11 are different and must be handled differently. In order to handle these data elements correctly during repository generation, a number of domain modeling concepts are identified by inspecting the domain model language, reviewing books [3] and internal documents, and through interviews with all stakeholders. These modeling concepts are described below.

3.3.1. DomainModel

The concept DomainModel represents the container for all other elements in the domain model. The instances of this DomainModel can be core or non-core. The core model contains common data elements that can be reused across multiple functions of the TWINSCAN. Non-core domain models contain data elements that are specific to a certain function. Non-core domain models can refer to the core domain model. However, core domain models can not refer to non-core domain models. Core and non-core models are further explained in Section 3.4 with examples.

3.3.2. Entities

Entities represent domain model elements that have a lifecycle and an identity, for example, wafers and lots. Every entity is considered to be unique and is identified by an ID. Entities with exactly the same attributes are considered to be different and are uniquely identifiable. This prevents confusing Entity instances with other Entity instances. For example, a particular physical wafer is always unique and should never be confused with another wafer. Properties of this wafer can change through time. However, the identity of this particular wafer continues to be the same. Data corruption is one of the severe consequences of mistaken IDs of Entity instances.
Entities are stored in their own repositories. In the domain model language, Entity is represented as Entity, as shown in Figure 8.

3.3.3. Mutability

Mutability is a property of Entities. Immutable Entities are Entities that can never be updated after creation. Mutable Entities are Entities that can be updated after creation.

The mutability of ValueObjects is determined by the mutability of the Entities they are part of. ValueObjects are considered to be immutable when they are part of immutable Entities. ValueObjects that are part of mutable Entities are considered to be mutable. Instances of the same ValueObject can have different mutability based on the mutability of the entity instances they are part of.

In the domain model language shown in Figure 8, mutability is defined as a property of Entities. This property is named as immutable and it can be true or false.

3.3.4. Volatility

Volatility is a property of Entities. Non-volatile Entities are Entities that survive the TWINSCAN system restart. Volatile Entities are Entities that do not survive the system restart.

The volatility of ValueObjects is determined by the volatility of the Entities they are part of. ValueObjects are considered to be volatile when they are part of volatile Entities. ValueObjects that are part of non-volatile Entities are considered to be non-volatile. Instances of the same ValueObject can have different volatility based on the volatility of the entity instances they are part of.

In the domain model language shown in Figure 8, volatility is defined as a property of Entities. This property is named as volatile and it can be true or false.

Non-volatile Entities can only be stored in a persistent repository. Volatile Entities can be stored in memory or persistent repositories.

3.3.5. ValueObjects

ValueObjects represent domain model elements that have no identity. ValueObjects with the same value are considered to be equal. A ValueObject is identified by its attributes. ValueObjects are used to describe parts of an entity. For example, in the domain model in Figure 11, the ValueObject Lotinfo is part of the entity Lot.

ValueObjects are not stored in their own repositories. They are stored together with the entity they are part of.

In the domain model language, ValueObjects are defined as specializations of structured elements.

3.3.6. Enumerations

Enumerations are used to specify a list of elements represented as enumeration literals. Enumerations are used to describe Entities and ValueObjects. In the domain model language, Enumerations are defined as specializations of Types.

3.3.7. PrimitiveTypes

PrimitiveTypes are used to specify the primitive data types that are used to define domain models. In the domain model language, primitive types are treated as ordinary Types.

3.3.8. Compositions and Attributes

Composition represents a whole/part relationship between elements in the domain model. In a composition relationship, the whole is also called a container. In this relationship, an instance of the part can only be contained in, at most, one instance of the container. If the container is deleted, the part is also deleted with it. However, the part can be deleted without deleting the container.

In the context of this work, compositions and attributes are considered to be equal. The part can be represented as an attribute of the container. In our domain, we only consider composition of ValueObjects. Entities and ValueObjects can contain ValueObjects.

3.3.9. Associations

Associations represent a unidirectional relationship between domain model elements. In our domain, we only consider associations towards Entities. Associations from an Entity or a ValueObject to an Entity are allowed. Associations towards ValueObjects are not allowed.

In the domain model language, associations are defined as TypedElements. Associations are contained by structured elements, i.e., Entities and ValueObjects.

3.3.10. Multiplicities

In the domain model language, we have three kinds of multiplicities:

i. Entity Multiplicity
   Entity Multiplicity is a property of Entities. It determines the number of entity instances that can be stored in a repository.

ii. Association Multiplicity
   Association Multiplicity is a property of associations. It is used to specify the allowed number of source and target instances involved in the association relationship.

iii. Compositions / attributes Multiplicity
   This multiplicity determines the allowed number of instances that can be contained by each instance of the container. The container can be an entity or a ValueObject.

Multiplicities are shown by using an interval of integers with a lower and an upper bound. In the domain model language, multiplicities are represented as Multiplicities, as shown in Figure 10. It is mandatory that multiplicities are explicitly specified.

3.3.11. MultiplicityConstants

MultiplicityConstants can be used to specify multiplicity ends. MultiplicityConstants should be given a value. Once defined, these MultiplicityConstants can be reused in multiple places. In the domain model language, these constants are represented as MultiplicityConstant.

3.3.12. Relationships between Entities and ValueObjects

The relationships between Entities and ValueObjects follow a number of rules. As an association points to something that must be identifiable, associations can only point to Entities. There was no reason to identify a part of a bigger whole. Therefore, Entities are disallowed to be contained. These rules are summarized, as shown in Table 1.

Table 1: Relationships between Entities and ValueObjects

Relationship Type	Composition	Association
VO1 → VO2	**YES** (Member variable)	**NO**
E → VO	**YES** (Member variable)	**NO**
VO → E	**NO**	**YES** (Navigability)
E1 → E2	**NO**	**YES** (Navigability)

3.4 *Core Domain Model*

A core domain model is a model of the core data aspects of the TWINSCAN system. Core domain models are owned by and can only be modified by ASML data architects. These models are stable models and are used across multiple ASML software components. Core models do not depend on non-core models. An example of such a core domain model is shown in Figure 11. This core domain model contains a number of data elements, namely Machine, Lot, Wafer, Chuck, LotInfo, ChuckEnum, and the Primitive Types such as Double and String, and the relationships between them. Each Lot belongs to a Machine and contains one LotInfo. Zero or more Wafers belong to a Lot. Zero or one wafer may be loaded on a Chuck for measurement or exposure. This implies that a Chuck may also be empty.

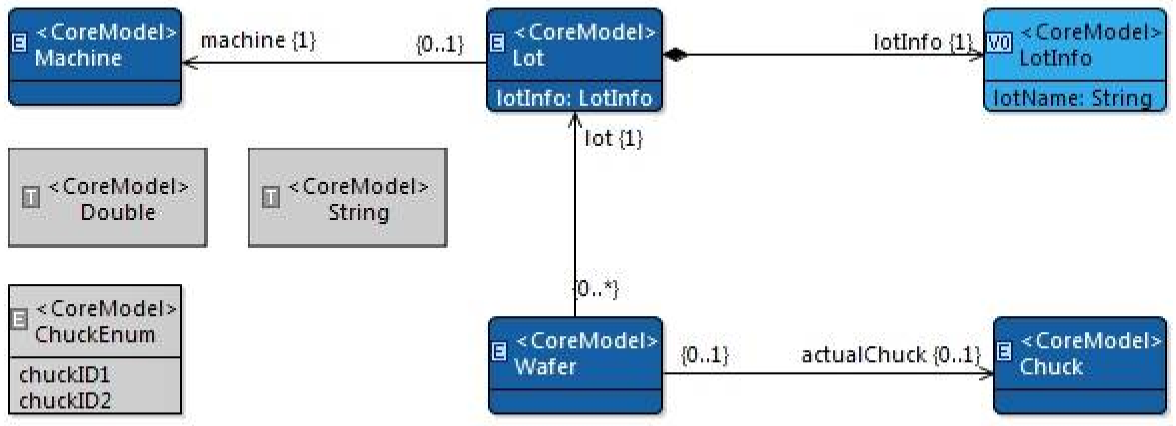


Figure 11: Example Core Domain Model

Other domain models reuse elements from the core domain model whenever applicable. For example, the domain model shown in Figure 12 reuses the element Machine and the Type String from the core domain model shown in Figure 11.

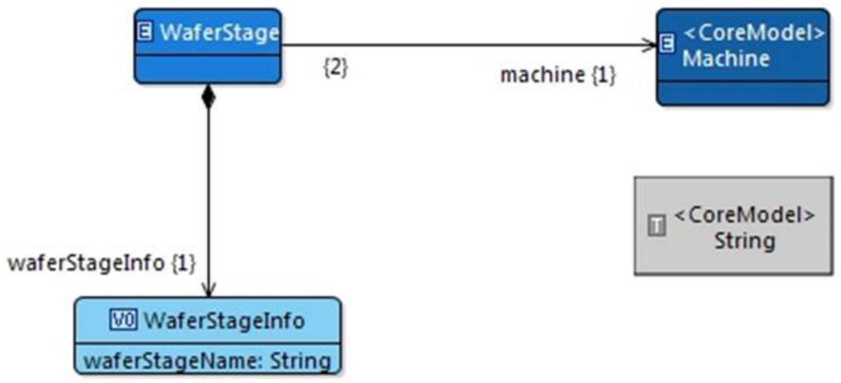


Figure 12: Example Extension, WaferStage Domain Model

3.5 *From domain models to generated repositories*

Domain models are specified in an implementation independent formalism by using the domain model language. The data as present on the scanner, as an instantiation of the domain model, needs to be stored in a repository. In order to reduce development time and increase efficiency, the implementation of these repositories is generated automatically by using the repository generation tool. The tool must provide a flexible way of configuring repository implementation choices without touching domain models.

A designer designs the domain model with the goal of generating code implementing the data concepts and generating code that implements repositories for Entities. This code then will be executed on the TWINSCAN system.

3.6 Implementation choices

Implementation choices allow generation of different flavors of repositories for different domain models. These concepts are: storage, orientation, communication, ID strategy, target identifier, target language, ASML SW component, Visibility, Target Path.

3.6.1. Storage

The storage concept answers the question ‘where to store Entity instances?’ All Entities in a domain model must be stored in a repository. This repository can be memory (boost implementation), database, or disk based. The concept storage allows the selection of one of these storage types. Depending on this choice, Entity instances are stored in the right repository.

3.6.2. Orientation

The concept orientation answers the question ‘how to access and update repositories containing Entities?’ with respect to orientations two classes of repositories are identified:

i. Clone – oriented repositories
These types of repositories provide explicit update operations to update entity instances. Clients work on a local clone of the Entities in the repository. Clients of these types of repositories do not see each other’s changes to the local clones of Entities. Changes are visible only when they are updated in the repository. These types of repositories are applicable to both in memory and on disk repositories. Since clients clone Entities from repositories and update changes in the repository, these repositories are less efficient with respect to execution.

ii. Reference – oriented repositories
These types of repositories do not provide an explicit update operation. Instead, clients operate directly on the entity by referring to it by its ID. Any modification is directly performed on the instance present in the repository. Clients of Entity instances stored in these types of repositories see each other’s changes instantly. These repositories are applicable to in memory storage. They are not practical for databases and disk based repositories.

3.6.3. Communication

The concept Communication provides the possibility to select whether an entity must be stored in intraprocess/local or interprocess/shared repositories.

i. Intraprocess repositories
These types of repositories are stored in a local memory. Intra-process repositories can only be accessed from within the same process that actually creates/opens them.

ii. Interprocess repositories
These types of repositories are stored in a shared memory.

3.6.4. ID Strategy

Entities are domain model elements that have a unique identity. The concept ID Strategy is used to configure the implementation of the concepts that identify Entities. This unique identity can be realized by using one of uuid, increasing integer, random string, or random number. Multiple ID strategies are necessary because the chosen ID strategy might affect the performance. For example, searching elements in a database by their UUID is far less efficient than using an incremented integer. However, incremented integers are much harder to keep unique over multiple executions.

3.6.5. Target Identifier

This concept is used to specify a preferred target identifier for a Type in the domain model. This target identifier can be new or from a legacy code. This concept is used during repository generation. For example, if the domain model contains the type Double, it is necessary to identify what this Double type corresponds to in the target implementation language during code generation. It might also be necessary to import legacy header files in order to use the target type. This target identifier is used to specify these targets.

3.6.6. ASML SW Component

This concept is used to specify target ASML software component, which will be used to store the repository implementation during code generation.

3.6.7. Visibility

The concept visibility provides a flexible way of specifying whether a model and the corresponding generated code is visible outside of the ASML software component or not.

3.6.8. Target Path

The concept targetPath provides a flexible choice of where to store the generated artifacts with respect to the location of the domain model.

3.6.9. Target Language

This concept is used to specify the target implementation language and the extension of header files. This concept provides two options: C++ and python.

3.7 Repository Interface Semantics

Depending on the choices of the implementation specific concepts and decisions made at the domain modeling level, different repositories are needed. This is illustrated below with an example for clone-oriented repositories. The Entities and ValueObjects of clone-oriented repositories must provide the following interfaces:

- Getters for the EntityId (valid only for Entities)
  - Returns own ID
- Getters for all attributes i.e. ValueObjects and Types
  - Return const reference
- Getters for all associations
  - Return the ID
- Setters for all attributes
  - Take const reference
- Setters for all associations
  - Take an ID

These interfaces are illustrated with an example for the Entity Lot, as shown in Figure 13. The Entity Lot is represented in the core domain model in Figure 11.

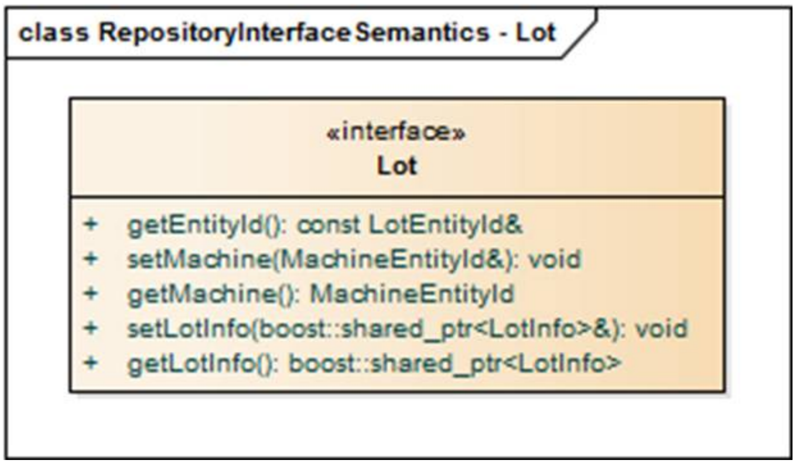

class RepositoryInterfaceSemantics - Lot
«interface»
Lot
+ getEntityId(): const LotEntityId&
+ setMachine(MachineEntityId&): void
+ getMachine(): MachineEntityId
+ setLotInfo(boost::shared_ptr<LotInfo>&): void
+ getLotInfo(): boost::shared_ptr<LotInfo>

Figure 13: Lot Entity Interfaces

Entity clones are independent of each other. Changes to a local clone of the Entity do not influence other clones. Updating an Entity clone to a repository does not change the contents of other clones. Removing an Entity from a repository does not change the contents of all clones of the Entity. It is possible to have multiple clones of the same Entity with different contents.

These repositories must provide interfaces for:

- Adding a new Entity instance
- Updating an existing Entity instance
- Getting a clone of an existing Entity instance based on ID
- Getting a clone based on an ID of an associated Entity instance
- Removing an Entity instance based on an ID

These interfaces are illustrated with an example for the Entity Lot, as shown in Figure 14. The Entity Lot is present in the core domain model in Figure 11.

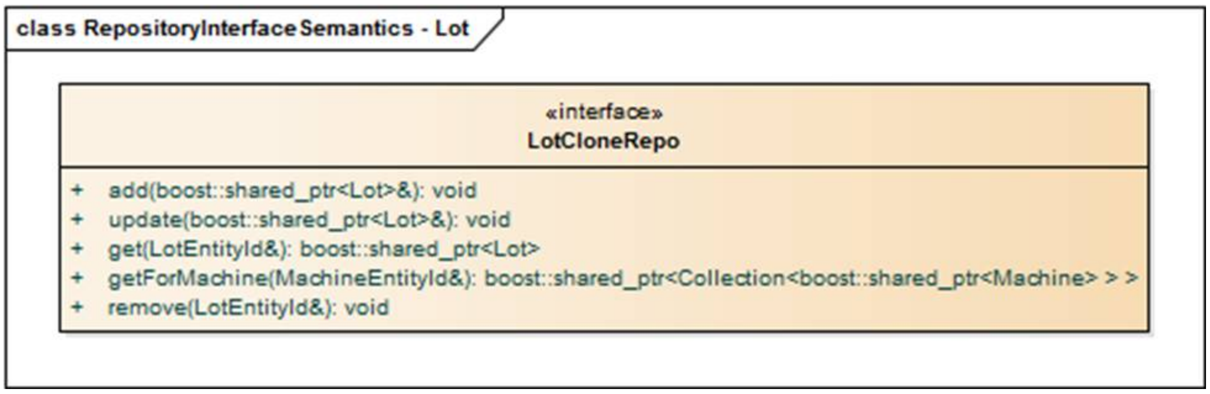

class RepositoryInterfaceSemantics - Lot
«interface»
LotCloneRepo
+ add(boost::shared_ptr<Lot>&): void
+ update(boost::shared_ptr<Lot>&): void
+ get(LotEntityId&): boost::shared_ptr<Lot>
+ getForMachine(MachineEntityId&): boost::shared_ptr<Collection<boost::shared_ptr<Machine> > >
+ remove(LotEntityId&): void

Figure 14: Interfaces for clone oriented repositories.

3.8 *Repository Implementation*

To demonstrate the repository implementation that must be generated by the repository generation tool, example implementation classes are given for the intraprocess/heap and interprocess communication options. The implementation classes in Figure 15 show the heap based repository implementation that must be generated for the Entity Wafer. The Entity Wafer is present in the core domain model in Figure 11.

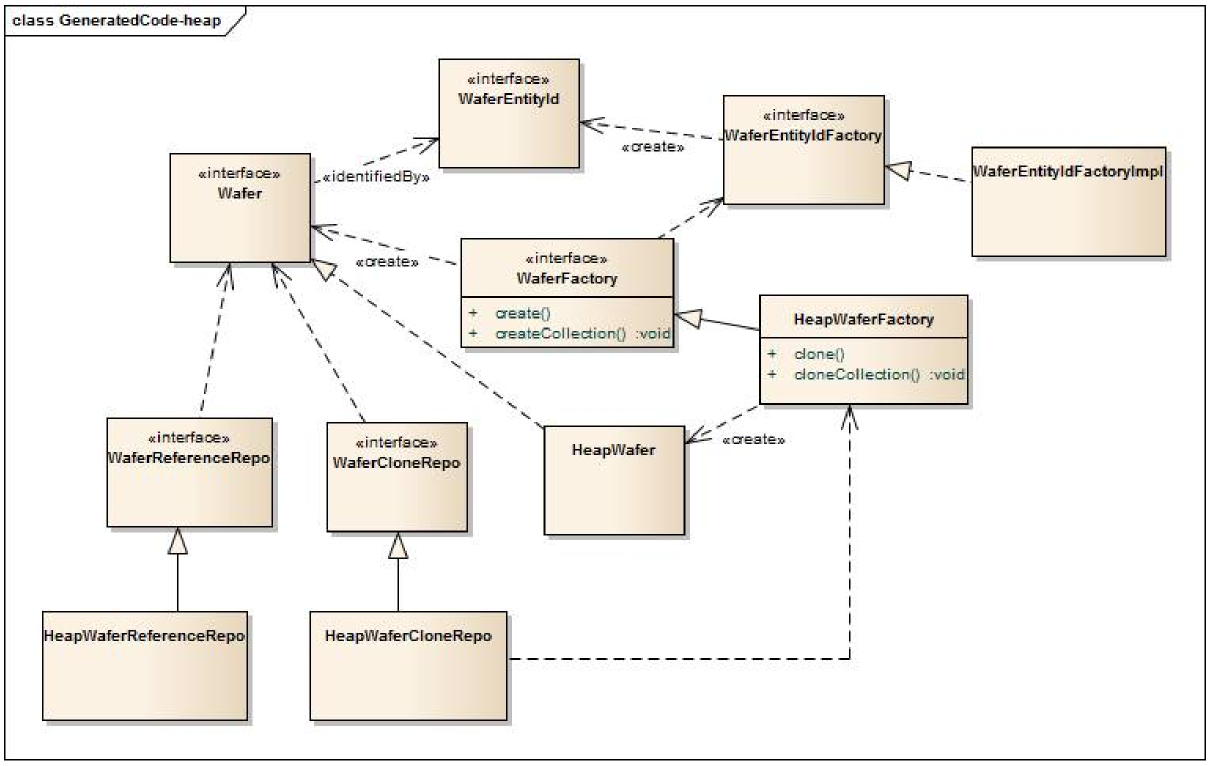


Figure 15: Heap implementation classes of the code that must be generated for the Entity 'Wafer'

The implementation classes in Figure 16 show the repository implementation that must be generated with the interprocess communication option selected for the Entity Wafer.

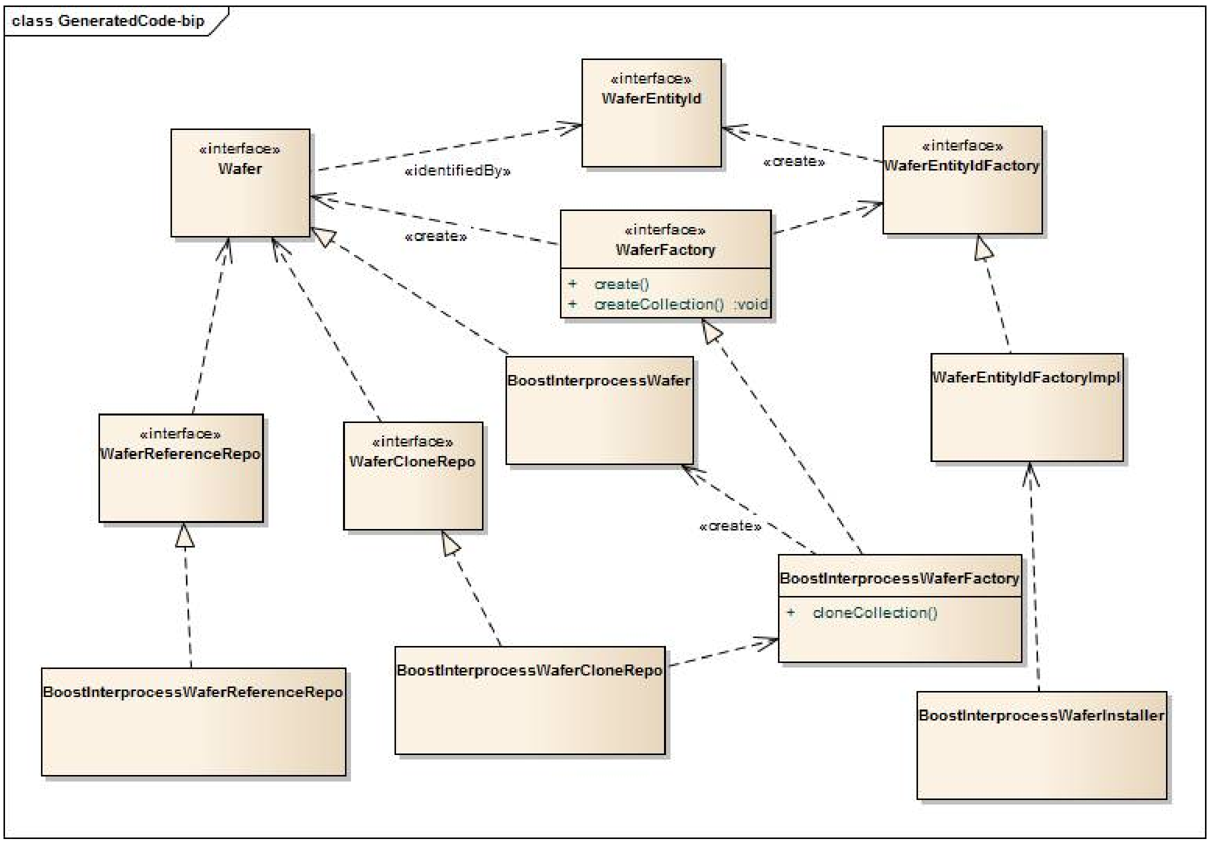


Figure 16: Boost interprocess implementation classes that must be generated for the Entity 'Wafer'

4.System Requirements

The problem and a thorough domain analysis are presented in Chapters 2 and 3 consecutively. The purpose of this chapter is to present the requirements considered for this project, their priority, and the main use cases derived from these requirements.

4.1 MoSCoW

MoSCoW [4] is a requirement prioritization technique containing the following levels:

1. Must Have (M) Requirements
   The requirements under this category must be satisfied for the product to be accepted.

2. Should Have (S) Requirements
   The requirements under this category should be satisfied if possible. It is not acceptable that all of the requirements in this category are completely ignored.

3. Could Have (C) / Nice to Have Requirements
   The requirements under this category could be satisfied if time and resources are available. The requirements under this category are referred to as Nice to have requirements in the rest of this report.

4. Won't Have (W) Requirements
   The requirements under this category will not be satisfied in the scope of this project. However, since they will be considered in the future, they can influence the design.

4.2 Requirements for the repository generation tool

The requirements for the repository generation tool were collected through interviewing with all stakeholders, brainstorming during weekly meetings with stakeholder and supervisors, analyzing existing documents, and prototyping.

Together with stakeholders and supervisors, the identified requirements were prioritized by using a suitable subset of the MoSCoW technique. Although they are realized differently, all functional and nonfunctional requirements and constraints were prioritized according to their importance regardless of their category. The Must Have, Nice to have, and Won't have requirement levels of MoSCoW are selected for this project. Although the Won't Have requirements will not be satisfied in the scope of this project, the provided solution architecture and design should not prohibit realization of these requirements in the future.

The Must Have and Nice to have requirements for the repository generation tool are described in detail based on the ASML EPS document format. The rationale behind each of these requirements is given. The test strategies that were used for testing each of these requirements are also explained.

4.2.1. Must Have Requirements (MReq)

The Must Have requirements regarding the Implementation Model Language and code generation are described in Table 2.

Table 2: Must Have Requirements (MReq)

ID	Description, Rationale, and Testing	Ref.
MReq 1	*Description:* The repository generation tool must clearly separate domain model and implementation model concepts. It must be possible to develop domain models without polluting these models with implementation and technology details. *Rationale*: If the domain and implementation model concepts are not separated, domain models would be highly coupled to specific implementation technology. This would make changing implementation technology without changing domain models impossible.	ASML Architecture Group & ASML Metrology Group

	Testing: This is tested by reviewing the design. The test passes if no implementation specific concepts are added to the domain model language; it fails otherwise.	
MReq 2	*Description:* The tool must support dependencies between different implementation models. It should be possible to refer to one model from another. In this situation the referred model must stay unchanged. *Rationale*: without this feature, it would not be possible to reuse existing implementation models. *Testing*: This is tested by reusing an existing implementation model while creating another model. It was possible to reference from the new model to the existing model. The referred model also stayed unchanged. This is also tested through reviews by supervisors and stakeholders.	ASML Architecture Group & ASML Metrology Group
MReq 3	*Description:* The implementation model language must support selection of implementation choices regarding the concepts: **MReq 3.1:** Storage. **MReq 3.2:** Orientation. **MReq 3.3:** Communication. **MReq 3.4:** Target Identifier. **MReq 3.5:** ID Strategy. **MReq 3.6:** ASML SW Component. **MReq 3.7:** Visibility. **MReq 3.8:** Target Path. **MReq 3.9:** Target Language. The detailed description of these concepts can be found in Section 3.6 of Chapter 3. *Rationale*: Without the ability to easily change these settings, it would not be possible to make implementation choices for repository generation without touching the domain model. *Testing*: This is tested by creating implementation models based on a domain. The ability to create instances of each of these concepts is also tested. This is also tested through reviews by supervisors and stakeholders.	ASML Architecture Group & ASML Metrology Group
MReq 4	*Description:* The tool must provide a textual editor for creating implementation models. *Rationale*: For usability reasons, such as ease of use, convenience, and the ability to easily compare and merge model instances, stakeholders preferred a textual editor over a graphical one. *Testing*: This is tested by creating several implementation models for several domain models model by using this textual editor. This is also tested through reviews by supervisors and stakeholders.	ASML Architecture Group & ASML Metrology Group
MReq 5	*Description:* The tool must provide a wizard for generating default implementation models. These implementation models must be directly usable without modification. *Rationale*: Without this wizard, learning and getting started with the tool would not be easy. *Testing*: This feature is tested by using the wizard to generate implementation models for a domain model. This is also tested through	ASML Architecture Group

	reviews by supervisors and stakeholders.	
MReq 6	*Description:* The implementation model language must support sharing common settings between different implementation models with a minimal modification. *Rationale*: Without this feature, creating implementation models would be time consuming and could involve writing more lines of code than required. *Testing*: This is tested by creating multiple implementation models and reusing model fragments to avoid repeating information in multiple models as much as possible. This is also tested through reviews by supervisors and stakeholders.	ASML Architecture Group & ASML Metrology Group
MReq 7	*Description:* the tool must support code generation for boost intraprocess (heap or local memory) clone-oriented repositories. *Rationale*: These kinds of repositories are required by another project within the Metrology group. *Testing*: This is tested by generating boost intraprocess clone-oriented repositories for a domain model. The generated code has built successfully. The correctness of the generated code is tested by using unit tests. The correctness of design of generators and languages are also tested through reviews by supervisors and stakeholders.	ASML Architecture Group & ASML Metrology Group
MReq 8	*Description:* the tool must support code generation for boost interprocess (shared memory) clone-oriented repositories. *Rationale*: These kinds of repositories are required by another project within the Metrology group. *Testing*: This is tested by generating boost interprocess clone-oriented repositories for a domain model. The generated code has built successfully. The correctness of design of generators and languages are also tested through reviews by supervisors and stakeholders.	ASML Architecture Group & ASML Metrology Group
MReq 9	*Description:* Using the implementation models with the existing version management tools must be easily possible. *Rationale*: Without this feature, it would not be possible to store and manage implementation models by using the existing version management tools. *Testing*: This is tested by creating an implementation model and storing them in a GIT master branch. A new branch is created and the implementation model is modified. It was possible to view the differences and merge the new branch with the master branch.	ASML Architecture Group
MReq 10	*Description:* It must be possible to use the tool standalone on an ASML computer. *Rationale*: ASML software architects want to use the tool standalone on an ASML computer. For example, the data architect uses the tool standalone to create implementation models for core domain models and generate code from them. If the tool cannot be used standalone on an ASML computer, these architects will not be able to use it. Furthermore, the tool will be used to analyze the system's behavior by people who are not using the ASML SW development environment. Therefore, the tool as a whole must be runnable outside of the ASML software development environment.	ASML Architecture Group

	Testing: This is tested by using the tool standalone on an ASML computer.	
MReq 11	*Description:* The solution must be scalable against the number of elements in a domain model. It must support repository generation from a domain model with at least 100 Entities and 10 model imports. *Rationale*: It is expected that models are of this size when applied within the TWINSCAN system architecture. *Testing*: This is tested by creating a domain model with 100 Entities and 10 model imports and generating a repository from this model.	ASML Architecture Group
MReq 12	*Description:* the tool must support validation of implementation models against several validation rules defined by ASML data specialists and software engineers before repository generation. *Rationale*: Without this feature, it would not be possible to identify invalid models that will result in generating code that does not build, or worse, code that builds but produces incorrect or unexpected results when deployed in the TWINSCAN system. *Testing*: This is tested by validating a set of valid and invalid models. Overviews of the validation errors that are automatically detected are shown.	ASML Architecture Group
MReq 13	*Description:* Regression tests must be provided to ensure that generated repository implementation via the new version of the tool does not break the functionality provided in the previous version. *Rationale*: Without this feature, it would not be possible to identify problems introduced during the evolution of the repository generation tool. The behavior we want to test is not whether the generator is deterministic, but if changes in the generation process do not lead to unwanted effects in the generated code. *Testing*: This is tested by running regression tests after repositories are generated from domain models via a new version of the repository generation tool.	ASML Architecture Group & ASML Metrology Group
MReq 14	*Description:* The solution must be based on open source technologies. *Rationale*: The tool is deployed in the Eclipse-based WindRiver Workbench. If the solution would be based on proprietary technologies, there would be a possibility of vendor lock-in and unnecessary software costs. *Testing*: This choice of technologies is discussed with supervisors and with the ASML Software Development Environment Group.	ASML Software Development Environment Group

4.2.2. Nice to Have Requirements (NReq)

The Nice to have requirements regarding implementation models and repository generation are described below:

Table 3: Nice to Have Requirements (NReq)

ID	Description, Rationale, and Testing	Ref.
NReq 1	*Description:* the tool must support code generation for Boost Intraprocess (Heap or local memory) reference-oriented repositories. *Rationale*: These kinds of repositories are required by another project within the Metrology group.	ASML Architecture Group &

	Testing: This is tested by generating Boost Intraprocess reference-oriented repositories for a domain model. The correctness of the generated code is tested by using unit tests.	ASML Metrology Group
NReq 2	*Description:* the tool must support code generation for Boost Interprocess (Shared Memory) reference-oriented repositories. *Rationale*: These kinds of repositories are required by another project within the Metrology group. *Testing*: This is tested by Boost Interprocess reference-oriented repositories for a domain model. The correctness of the generated code is tested by using unit tests	ASML Architecture Group & ASML Metrology Group
NReq 3	*Description:* The tool must be deployable in the ASML's Eclipse-based WindRiver Workbench for Linux environment. Since ASML is moving towards the Eclipse Luna, the tool must be based on the Luna version of Eclipse. *Rationale*: ASML software architects and software engineers use the Eclipse-based WindRiver Workbench as a development environment. If the tool cannot be deployed in the WindRiver Workbench, it will not be handy to be used by these architects and software engineers. *Testing*: Since the ASML WindRiver workbench does not yet support Eclipse Luna, the deployability of the tool is tested in an Eclipse Luna in a standalone ASML computer. The tool will also be sent to the ASML Software Development Environment Group for testing.	ASML Architecture Group & ASML Software Development Environment Group
NReq 4	*Description:* The tool must support relating the generated code to the version of the repository generation tool, the domain model, and implementation models. *Rationale*: Without this feature, it may not be easy to trace the versions of domain and implementation models, version of the tool from which this code is generated during maintenance and diagnostics. *Testing*: This is tested by manually checking the generated code. The test passes if the generated code contains information about the versions of the domain model, the repository generation tool, and the implementation model, it fails otherwise.	ASML Metrology Group

4.2.3. Won't have requirements (WReq)

Table 4: Won't have Requirements (WReq)

ID	Description, Rationale, and Testing	Ref.
WReq 1	*Description:* the tool supports C++ code generation for database repositories. *Rationale*: Without this it will not be possible to store non-volatile Entities in database repositories. *Testing*: This is tested by generating a database repository implementation for a domain model containing non-volatile Entities. The correctness of the generated code is tested by using unit tests.	ASML Architecture Group & ASML Metrology Group

WReq 2	*Description:* the tool supports C++ code generation for disk based repositories. *Rationale*: Without this it will not be possible to store non-volatile Entities in disk based repositories. *Testing*: This is tested by generating a disk based repository implementation for a domain model containing non-volatile Entities. The correctness of the generated code is tested by using unit tests.	ASML Architecture Group & ASML Metrology Group
WReq 3	*Description:* the tool supports python code generation. *Rationale*: Without this it will not be possible to generate python repositories from domain models. *Testing*: This is tested by generating a python repository implementation for a core domain model. The correctness of the generated code is tested by using unit tests.	ASML Architecture Group & ASML Metrology Group

4.3 Use cases

The main interactions between the primary actors and the repository generation tool are described by using the use case diagram shown in Figure 17. The primary actors are identified from the stakeholders discussed in Chapter 2. These primary actors are stakeholders that directly interact with the system and initiate a service to accomplish a certain goal. The user actor represents the ASML data architects and software engineers. Since the activities of the data architects and software engineers do not differ, they are both mapped to the generic user actor.

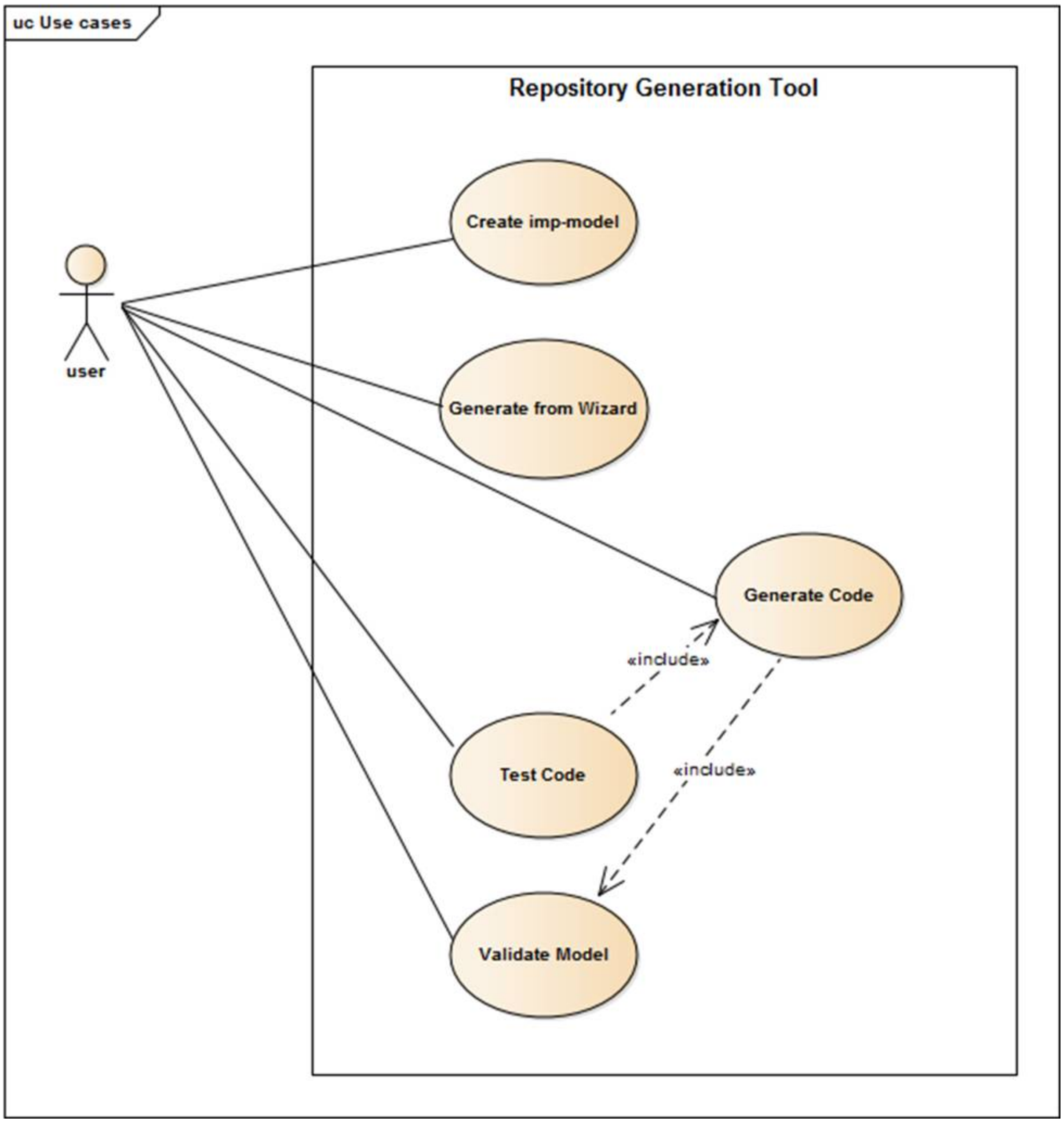


Figure 17: The main use cases, showing the interactions of primary actors with the repository generation tool.

The order in which these use cases are executed is described by using the activity diagram in Figure 18.

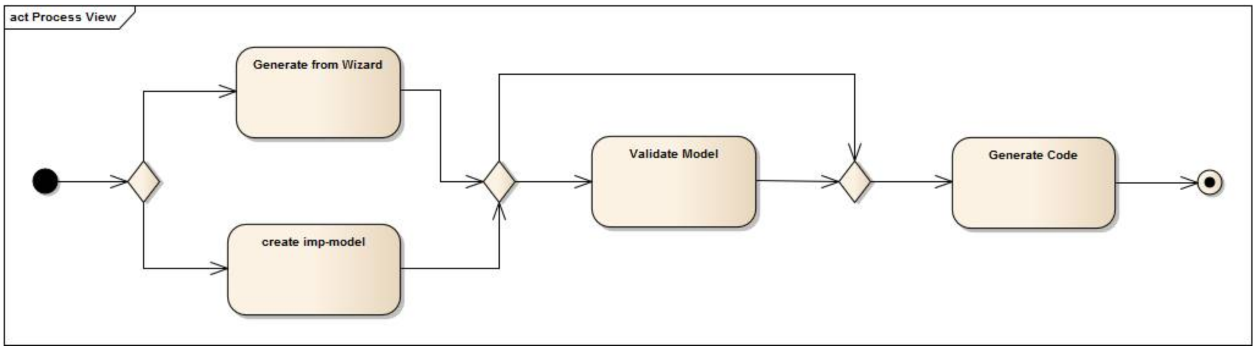


Figure 18: Process View, showing the order in which the use cases of the repository generation tool are executed.

4.4 *Use case description*

The detailed description of the use cases introduced in the previous Section is given below. These detailed use case descriptions are written using the templates defined by Cockburn [5].

4.4.1. Create imp-model use case

Use Case	**RGT1**
Name:	Create imp-model
Scope:	Repository Generation Tool (RGT)
Level:	User goal
Primary Actor:	User (Data Architect (DA) or Software Engineer (SE))
Stakeholders & Interests:	DA – wants to define implementation models for core domain models. SE – wants to define implementation models for extension domain models.
Precondition:	domain model exists.
Minimal Guarantees:	Users are able to generate default implementation model from their domain model through a wizard.
Success Guarantees:	DA/SE has created imp-model.

Main Success Scenario:

1. User selects storage kind.
2. User selects orientation kind.
3. User selects communication.
4. User selects ID strategy.
5. User selects target implementation language.
6. User selects target ASML component.
7. User selects visibility for the generated code.
8. User selects the domain model elements for which this implementation model is applicable.

Extensions:

1a – 8a. User does not know how to create implementation models:
    1a1 – 8a1. RGT provides a wizard from which a default implementation model can be generated.
1a – 7a. User wants to use settings from existing implementation models.
    1a1 – 7a1. RGT provides a means to reuse these settings.

4.4.2. Generate from Wizard use case

Use Case	**RGT2**
Name:	Generate from Wizard
Scope:	Repository Generation Tool (RGT)
Level:	User goal
Primary Actor:	User (Data Architect (DA) or Software Engineer (SE))
Stakeholders & Interests:	DA – wants to generate default imp-model from core domain model. SE – wants to generate default imp-model from extension domain model.
Precondition:	domain model exists.
Minimal Guarantees:	RGT warns when users try to overwrite existing domain models.
Success Guarantees:	DA/SE has generated default implementation model.

Main Success Scenario:

1. User selects domain model.
2. User requests default imp-model generation through a wizard.
3. RGT checks if imp-model with the default name exists already.
4. RGT presents warning messages if imp-models already exist.
5. User decides to generate the default imp-model.
6. RGT generates the default imp-model from the domain model.

Extensions:

1a. User does not want to overwrite existing imp-models:
    1a1. RGT gives users the ability to quit the imp-model generation.

4.4.3. Generate Code use case

Use Case	**RGT3**
Name:	Generate Code
Scope:	Repository Generation Tool (RGT)
Level:	User goal
Primary Actor:	User (Data Architect (DA) or Software Engineer (SE))
Stakeholders & Interests:	DA – wants to generate code from core domain model based on imp-model for this core domain model. SE – wants to generate code from extension domain model based on imp-model.
Precondition:	imp-model and its corresponding domain model exist.
Minimal Guarantees:	RGT – does not generate code for invalid models RGT – informs the user that models are invalid.
Success Guarantees:	DA/SE has generated code from domain models based on their imp-models.

Main Success Scenario:

1. User requests code generation.
2. RGT checks if models are valid.
3. RGT generates code.

Extensions:

2a. RGT detects invalid models
    2a1. RGT reports about violated rules to the user.
    2a2. User quits code generation or corrects his models.

4.4.4. Validate Model use case

Use Case	**RGT4**
Name:	Validate Model
Scope:	Repository Generation Tool (RGT)
Level:	User goal
Primary Actor:	User (Data Architect (DA) or Software Engineer (SE)), RGT
Stakeholders & Interests:	DA – wants to validate the imp-model for his core domain models. SE – wants to validate the imp-models for his extension domain models. RGT – wants to check validity of models before code generation.
Precondition:	imp-model and its corresponding domain model exist.
Minimal Guarantees:	RGT – reports violated rules to the user.
Success Guarantees:	DA/SE has validated his model.

Main Success Scenario:

1. User/RGT requests model validation.
2. RGT checks model validity.
3. RGT reports violated rules to the user.

Extensions:

None

4.4.5. Test Code use case

Use Case	**RGT4**
Name:	Test Code
Scope:	Repository Generation Tool (RGT)
Level:	User goal
Primary Actor:	User (Data Architect (DA) or Software Engineer (SE))
Stakeholders & Interests:	User – wants to check if previous features are not broken after code generation with a new version of the RGT.
Precondition:	Repository generation tool has changed.
Minimal Guarantees:	RGT – reports the results of this check to the users.
Success Guarantees:	DA/SE has checked the new generation does not break previous features.

Main Success Scenario:

1. User requests regression test run.
2. RGT runs tests.
3. RGT reports results to the user.

Extensions:

None

5.System Architecture

The requirements for the repository generation tool are discussed in Chapter 4. The purpose of this chapter is to describe the high level architecture of the repository generation tool and document the architectural tradeoffs made in this project.

5.1 *High Level Architecture*

The Repository Generation Tool enables users to generate repository implementation from domain models based on choices in implementation models. The reference architecture of the Repository Generation Tool consists of several components that realize the different functionalities that it provides. The dependencies between these components are shown in the architecture. The details of these dependencies are discussed in Section 5.4 in the 4+1 Architectural Model [6]. The high level reference architecture is shown in Figure 19.

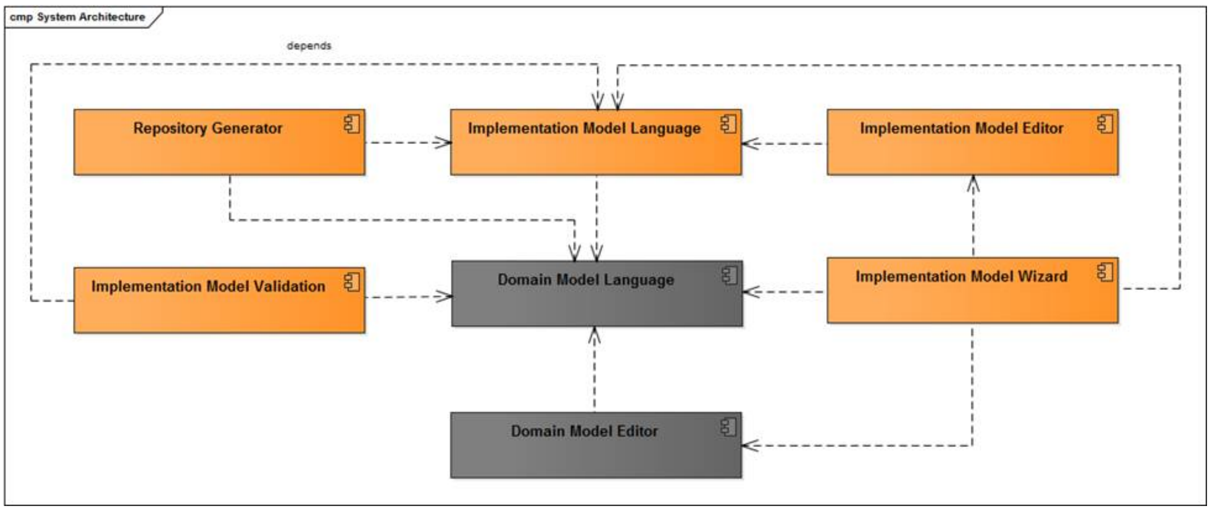

cmp System Architecture
depends
Repository Generator
Implementation Model Language
Implementation Model Editor
Implementation Model Validation
Domain Model Language
Implementation Model Wizard
Domain Model Editor

Figure 19: Reference Architecture, *showing the components that realize the required functionalities. The grayed out components are existing components.*

The different components in the reference architecture are briefly explained below.

1. Implementation Model Language
This component represents a domain specific language for defining implementation models. Implementation Models, imp-models for short, are the choices made by software engineers regarding repository implementation technology details and the domain model elements for which this implementation model is applicable. To allow referring to domain model elements from implementation models, the implementation model language depends on the domain model language.

2. Implementation Model Editor
This component represents a textual editor for implementation models. Since the syntax rules in the implementation model editor require metaclasses from the implementation model language, this component depends on the implementation model language.

3. Repository Generator
This component represents several modules containing various model-to-text transformation templates to enable users to generate repository implementations from domain models based on implementation models. To be able to process both implementation and domain models during code generation, this component depends on the implementation model and domain model languages.

4. Implementation Model Wizard
This component is a model-to-text transformation module to realize generation of initial implementation models from domain models for usability reasons. The generation of implementation models is from textual

and/or graphical representations of the domain model. This component depends on the implementation model language, implementation model editor, domain model language, and domain model editor.

5. Implementation Model Validation

This component represents an implementation model validation module to identify invalid implementation models when generating code. To be able to process both implementation and domain models during validation, this component depends on the implementation and domain model languages.

6. Domain Model Language

This component represents an existing domain specific language for defining domain models. This component does not depend on any other component in the reference architecture. Since the goal of the repository generation tool is generating code from domain models, all components in the reference architecture except the Implementation Model Editor depend on the domain model language.

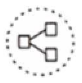
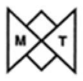
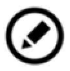
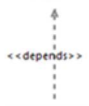
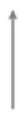
7. Domain Model Editor

This component represents existing textual and graphical editors for domain models. Since the syntax rules in the domain model editor require metaclasses from the domain model language, this component depends on the domain model language.

5.2 Architectural Notations

The structural and behavioral aspects of the architecture of the repository generation tool are described using a combination of UML and an ASML specific language called Language Modeling Language. The Language Modeling Language is used for modeling language architectures, dependencies, and model transformations within ASML. Since the language is under development, there is no literature reference for the notations used in this language. Therefore, the semantics of the notations that are used in the rest of this report are provided in Table 5.

Table 5: Language Modeling Language Notations that are used in the rest of this report.

No.	Notation	Semantics
1		This notation represents a language definition.
2		This notation represents a model-to-text transformation.
3		This notation represents an editor of a language.
4		As indicated by its stereotype, this notation represents language dependency.
5		The semantics of this notation is determined by its stereotype. If its stereotype is <<*edits*>>, it represents the relationship between a language and an editor. If no stereotype is specified, it represents input or output language definitions of a transformation.

5.3 *The Adapted MDA-based Approach*

Model Driven Architecture (MDA) is an approach to software systems in which the specification of system functionality is separated from the specification of the implementation on a specific technology [7]. In this way, MDA promises a long-term flexibility with respect to technology choices. To realize flexibility in our architecture by decoupling domain models from repository implementation technology details, two MDA-based approaches are considered. These approaches are: the MDA-based approach shown in Figure 20 and its adapted lightweight version shown in Figure 21. To help readers understand the correspondences between our models and the MDA models, a brief description is given below.

1. Domain Models
Domain models represent the data aspects of the TWINSCAN system. These models are purely about data regardless of repository implementation technology-specific details. These domain models correspond to the Platform Independent Models (PIM) of the MDA. Domain models conform to the Domain Model Language.

2. Implementation Models (imp-models)
Implementation Models, imp-models for short, are the choices made by software engineers regarding repository implementation technology details and the domain model elements for which this implementation model is applicable. The goal of these implementation models is twofold: it prevents pollution of the domain model with implementation details and it allows the generation of different repositories from the same domain model to facilitate storing and sharing of data at run-time. These models correspond to the platform specific information added to the PIMs while generating Platform Specific Models (PSM) in the Model-to-Model Transformation (M2M) step of the MDA. Implementation models conform to the Implementation Model Language.

3. Repository Generator
The Repository Generator performs the generation of a repository implementation from domain models. The Repository Generator corresponds to the Model-to-Text Transformation (M2T) step of the MDA.

In the MDA-based approach, the input to the Repository Generator is a PSM, as shown in Figure 20. PSMs are automatically derived from domain models and implementation models through a M2M. During this transformation step, the repository implementation-specific information contained in imp-models is incorporated into domain models. To realize the ability to refer to domain model elements from implementation models, we introduce a dependency between the Implementation Model and the Domain Model Languages.

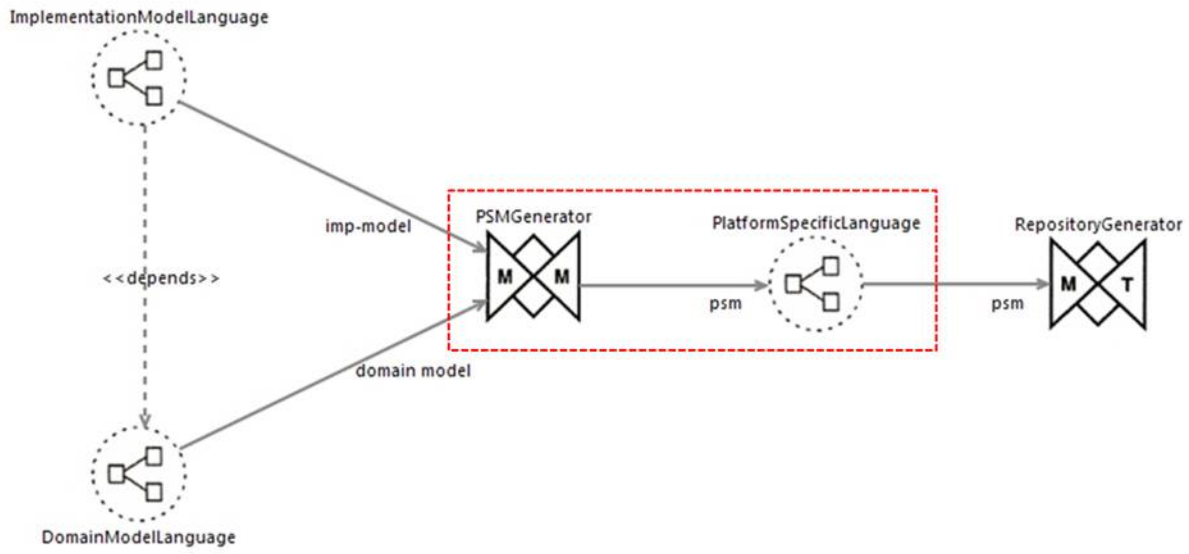


Figure 20: The MDA-based approach

In the adapted MDA-based approach, the implementation models and domain models are direct inputs to the Repository Generator, as shown in Figure 21.

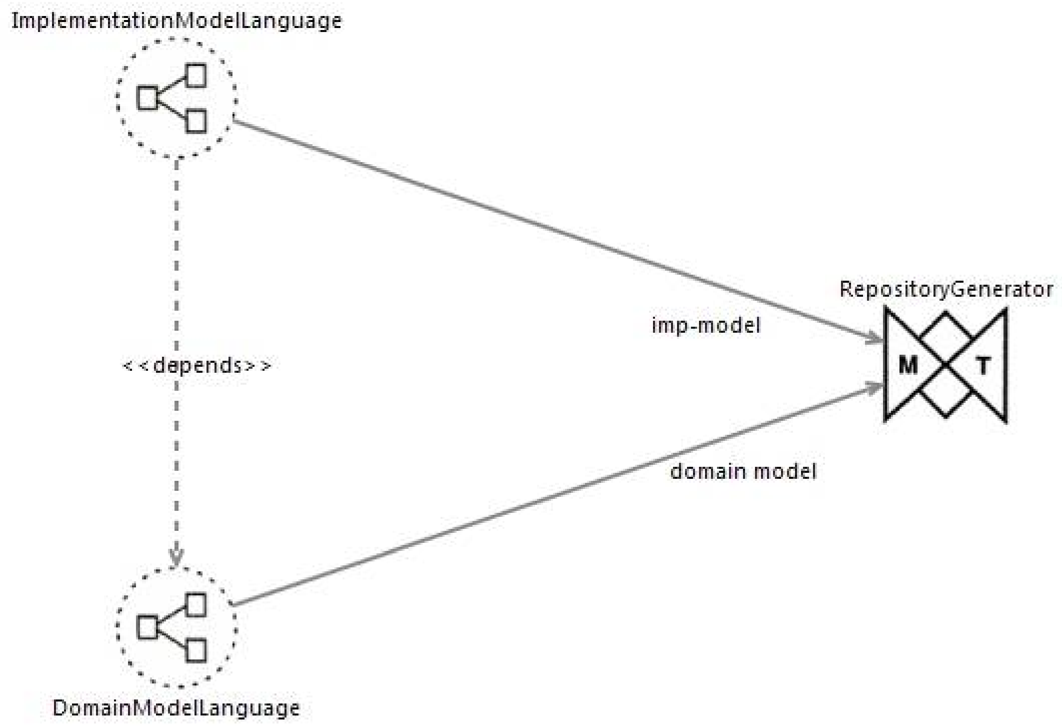


Figure 21: The Adapted MDA-based Approach

The main difference between the two approaches is the input models to the RepositoryGenerator. In the MDA-based approach, the PSM is the input to the RepositoryGenerator. In this approach, the PSMGenerator and PlastformSpecificLanguage definition are required to derive the PSM from domain models and imp-models. This approach is in line with the original MDA approach of the Object Management Group (OMG). In the adapted MDA-based approach, the domain models and imp-models are direct inputs to the RepositoryGenerator. Since we do not have PSMs in this approach, the PSMGenerator and PlatformSpecficLanguage definition that are necessary in the previous approach are not required. With respect to functionality, both approaches fit for our purpose.

Since the PSMGenerator and PlatformSpecificLanguage components are not required, the adapted MDA approach is lightweight. Furthermore, since a similar approach is being used in other projects within ASML, stakeholders preferred the adapted MDA approach over the MDA-based approach. Therefore, we selected the adapted MDA-based approach in this project.

5.4 The 4+1 Architectural Model

The 4+1 Architectural Model describes a system by using multiple views to separately address the concerns of multiple stakeholders [6]. The different aspects of the repository generation tool are described by using selected views of the 4+1 architectural model. The dependencies between the components in the reference architecture are described in the logical view. The deployment of the repository generation tool is described in the deployment view. Although the use case view of the 4+1 architectural model is relevant for this project, it is not described in this section. This is because the system level use cases are detailed in Chapter 4.

5.5 Logical View

In this view, the high level architecture of the repository generation tool is described by focusing on different parts of the architecture at a time. The dependencies between the Implementation Model Language, Domain Model Language, Repository Generator, Implementation Model Editor, Domain Model Editor, and Target Language are shown in Figure 22. The Implementation Model Editor edits the Implementation Model Language. To allow referring to domain model elements from implementation models, the Implementation Model Language depends on the Domain Model Language. The Repository Generator component takes implementation models that conform to the Implementation Model Language and domain models that conform to the Domain Model Language as an input. The Repository Generator produces repository code in the selected target implementation language.

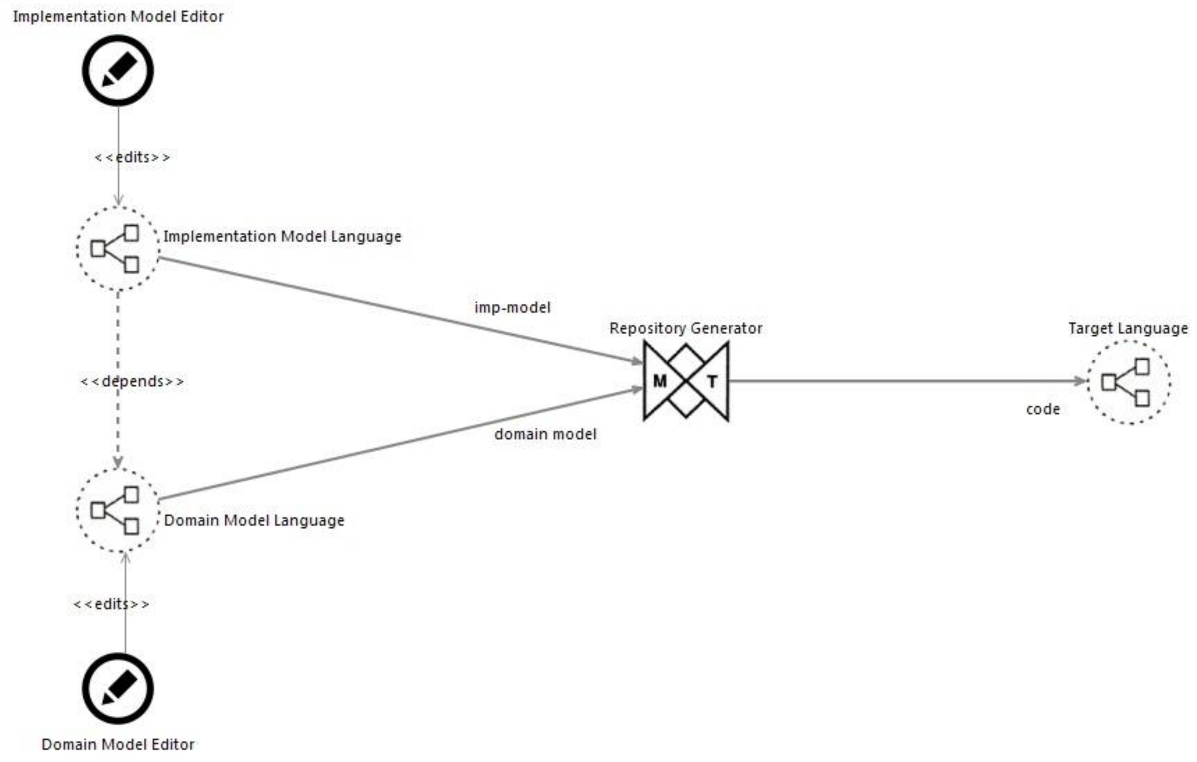


Figure 22: Logical View of the Repository Generation

The dependencies between the Implementation Model Wizard, Implementation Model Language, and Domain Model Language components are shown in Figure 23. The Implementation Model Wizard component takes domain models that conform to the Domain Model Language as an input and produces implementation models that conform to the Implementation Model Language.

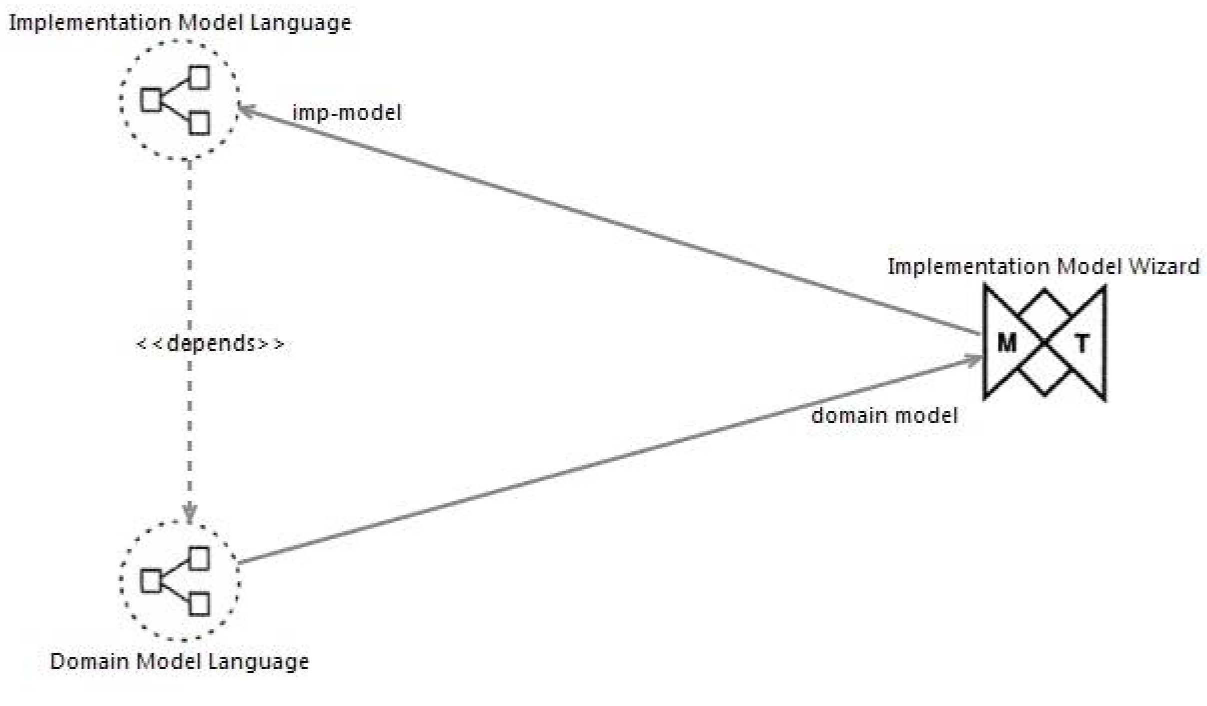


Figure 23: Logical View of the Implementation Model Wizard

The dependencies between the Implementation Model Validation, Implementation Model Language, and Domain Model Language components are shown in Figure 24. The Implementation Model Validation takes implementation models and domain models as an input and produces error messages that conform to the selected error reporting format.

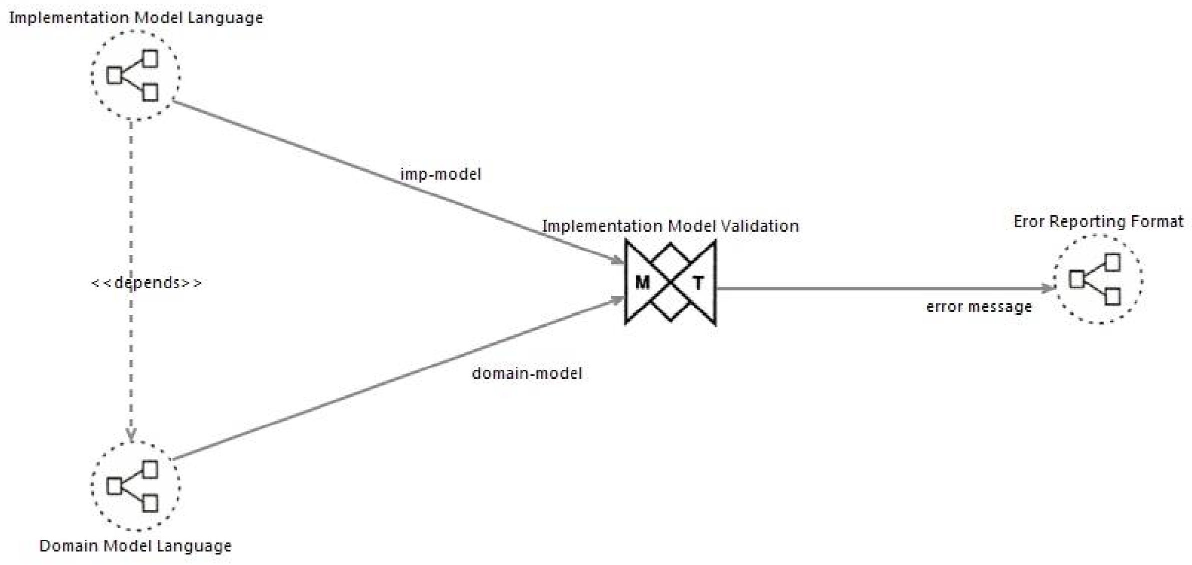


Figure 24: Logical View of the Implementation Model Validation

5.6 *Deployment View*

The repository generation tool is integrated with a bigger data modeling tool suite called a DCA Tool. The Domain Model Language and Domain Model Editor in which the repository generation tool depends on are part of the DCA Tool. The repository generation tool, together with the DCA Tool, is entirely deployed within eclipse. The Eclipse Modeling Framework provides a runtime environment to all of the components in these tools, as shown in Figure 25.

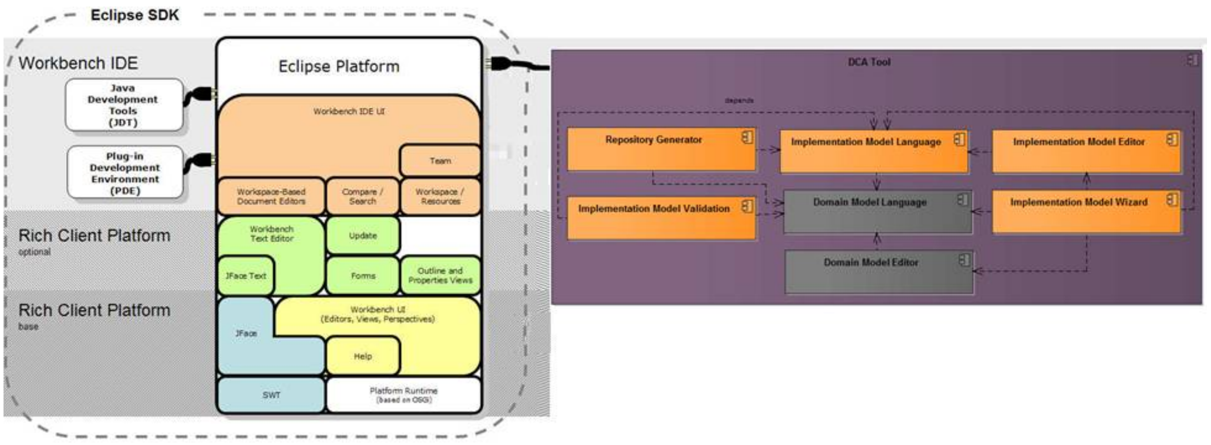

Eclipse SDK
Workbench IDE
Java Development Tools (JDT)
Plug-in Development Environment (PDE)
Rich Client Platform
optional
Rich Client Platform
base
Eclipse Platform
Workbench IDE UI
Team
Workspace-Based Document Editors
Compare / Search
Workspace / Resources
Workbench Text Editor
Update
JFace Text
Forms
Outline and Properties Views
Workbench UI (Editors, Views, Perspectives)
JFace
Help
SWT
Platform Runtime (based on OSGi)
DCA Tool
depends
Repository Generator
Implementation Model Language
Implementation Model Editor
Implementation Model Validation
Domain Model Language
Implementation Model Wizard
Domain Model Editor

Figure 25: Deployment view of the repository generation tool, together with the DCA Tool.

5.7 Architectural Principle

The core architectural principle behind the major functionalities of the Repository Generation Tool is the recipe-ingredient approach in a cookbook. This principle is briefly explained in this section. The detailed design of this architectural principle is presented in Chapter 6.

The architecture of the Repository Generator and Implementation Model Language components is inspired by the analogies of meals, recipes, and ingredients in a cookbook. The generated code resembles the meal to be cooked by the generator. For this, the generator uses a recipe represented by an implementation model. The recipe consists of ingredient that state details affecting the generation process. The ingredients can be reused in multiple recipes. So, configuration details can be reused over multiple recipes.

To maintain a high flexibility, the repository generator and the implementation model language components are designed to work based on the recipe-ingredient architectural principle. The implementation model language component deals with the creation of recipes and ingredients. Recipes specify the repository implementation details as ingredients and refer to the domain model elements for which code must be generated. The main ingredients of the implementation model language are the repository-implementation-specific concepts explained in Chapter 3. To generate the right repository from the domain model, the repository generator component requires an implementation model containing a recipe. This recipe combines the ingredients and refers to domain model elements. The referred domain model elements are elements for which this recipe is applicable. The repository generator generates the right repositories based on the recipes in the implementation model.

This recipe-ingredient based flexible architecture allows ASML data architects and software engineers to easily define specific recipes or change ingredients whenever different repositories are needed. The decision to model ingredients as a separate concept in the implementation model language is to allow reusing ingredients in multiple recipes.

5.7.1. Domain Model Elements

The domain model elements that are considered during repository generation are: Entities, ValueObjects, Enumerations, Types, and DomainModels. Code is generated for Entities, ValueObjects, Enumerations, and Types. Repositories are generated only for Entities. ValueObjects do not have dedicated repositories. They are stored together with the Entities they are part of. Each of these elements of domain models is explained in Chapter 3 as part of the domain analysis.

5.7.2. Ingredients

The main ingredients for repository generation are: storage, orientation, communication, ID strategy, target identifier, and target language. Each of these ingredients is explained in Chapter 3.

5.7.3. Recipes

Recipes combine selected ingredients and refer to domain model elements for repository generation. In this project we identify five Types of recipes. These recipes are: EntityRecipe, ValueObjectRecipe, EnumerationRecipe, TypeRecipe, and DomainModelRecipe. These recipes specify ingredients for each of the domain model elements they refer to.

5.8 Implementation Model Architecture

We have seen that an implementation model is composed of recipes and ingredients. In this section, we focus on the relationships of implementation models and domain models. The Language Modeling Language which is used to model language architectures within ASML does not allow modeling dependencies between model instances. Therefore, to help visualize the Implementation Model Architecture with examples, UML object diagrams are used. To tackle the scalability design challenge, the architecture allows definition of multiple implementation models of manageable size per domain model. For example, it is allowed to create three implementation model instances that correspond to a single domain model instance, as shown in Figure 26. In this way, the architecture can handle arbitrarily large domain models.

In general, an implementation model can be one of the following types:

1. Implementation model containing only ingredients, as shown in Appendix 2.

2. Implementation model containing only super recipes that can be extended by specific recipes, as shown in Appendix 3.
3. Implementation model containing recipes that can be used as a direct input for code generation, as shown in Appendix 4.
4. Implementation model containing combinations of the above which may be used for code generation.

It is up to the users to decide on how to organize their implementation models. However, the first and second types of models should never be used as a direct input for code generation. They should only be used to support code generation. The implementation models containing recipes for a domain model and its elements are used as a direct input for code generation.

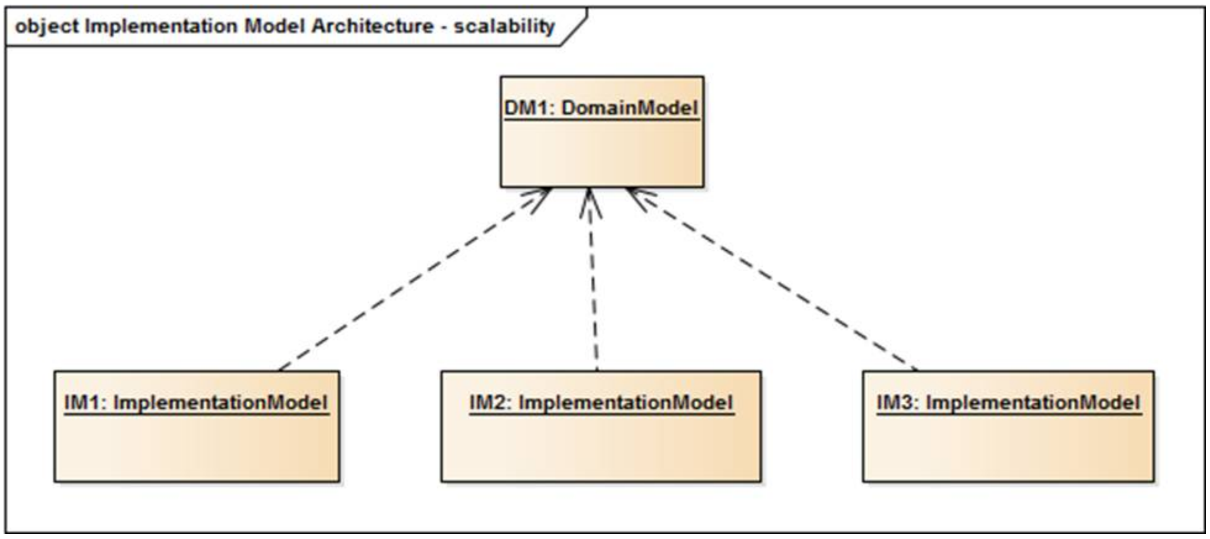


Figure 26: Implementation Model Architecture, Scalability

To be able to compose bigger models from smaller ones, a domain model can refer to elements of another domain model. For example, DomainModel1 imports DomainModel2, as shown in Figure 27. In this situation, the implementation model corresponding to DomainModel1 which is ImplementationModel1 must import the implementation model corresponding to DomainModel2 which is ImplementationModel2. This helps the code generator find code generation recipes for the referred elements of DomainModel2.

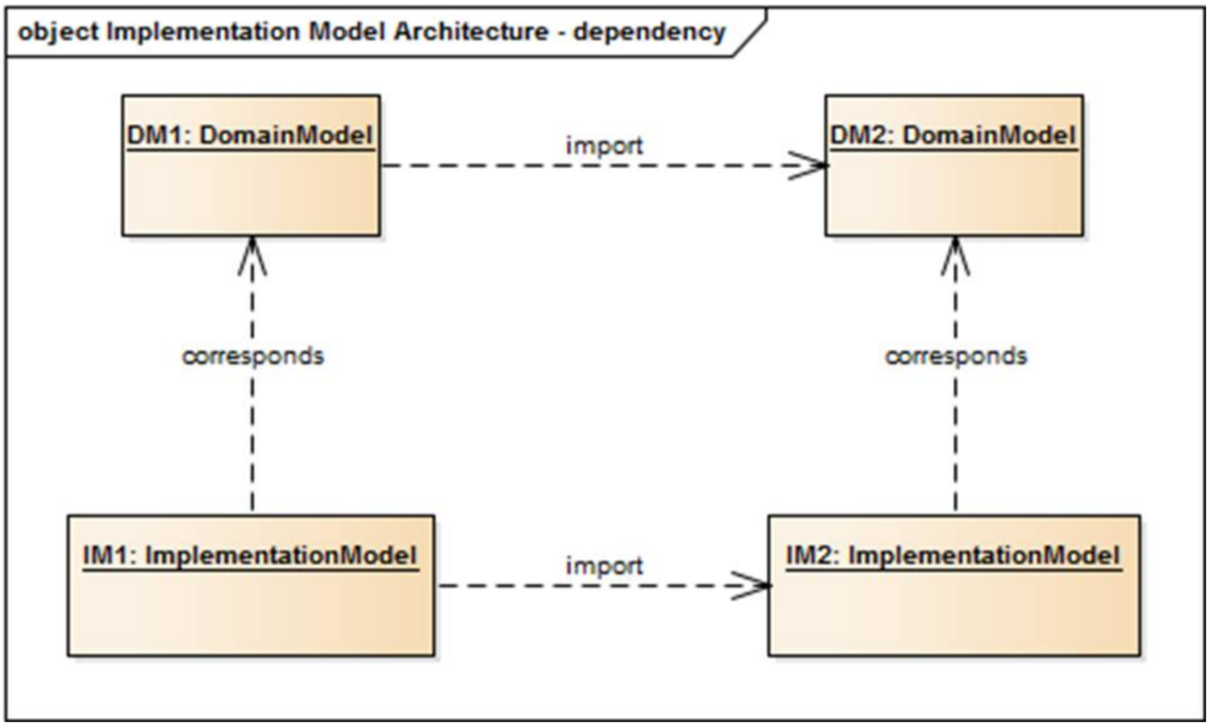


Figure 27: Implementation Model Architecture, dependency between imp-models and domain models

6.System Design

The high level architecture of the repository generation tool is discussed in Chapter 5. The purpose of this chapter is to discuss the detailed design of the components in the reference architecture. These components are: Implementation Model Language, Implementation Model Editor, Repository Generator, Implementation Model Wizard, and Implementation Model Validation. The tradeoff design decisions that guided the design process are also documented.

6.1 Implementation Model Language Design

The implementation model language is a domain specific modeling language used to specify implementation details of a domain model for repository generation. At the heart of the implementation model language design is the recipe-ingredient philosophy. Recipes are created by selecting appropriate ingredients and the domain model elements for which this recipe is applicable. These recipes guide the code generator in generating the right repositories from a domain model. To realize the implementation model language, various metamodels are defined. Since the repository generation tool must be based on open source technologies that are deployable in Eclipse, the Eclipse Modeling Framework (EMF) [8] is selected as the metamodeling framework in this project. EMF provides Ecore as the metamodeling language. Therefore, the various metamodels of the implementation model language are defined in Ecore.

Part of the metamodel of the implementation model language is shown in Figure 28. The concept ImplementationModel is the container of all implementation model elements. All concepts in the implementation model language are NamedElements. ImplementationModel contains zero or more Ingredients, Recipes, and Imports. The decision to model the multiplicities of the ingredient, recipe, and import associations as zero or more each is to allow creation of implementation model instances with only Ingredients or only Recipes. The concept Import allows importing domain models and implementation models into scope. The imported domain models are reused or referred to from implementation models while defining recipes. The ImplementationModel concept provides getAllRecipes() API to retrieve all recipes that are in the scope of the implementation model. This includes recipes that are imported.

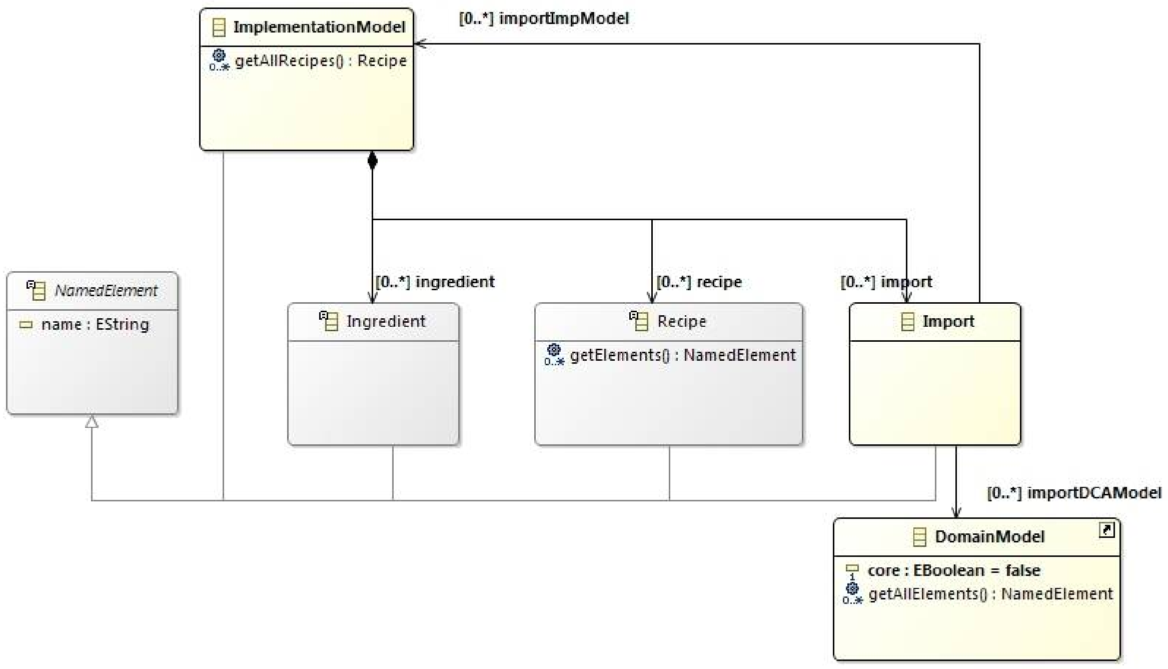


Figure 28: Implementation Model Language Metamodel

6.2 *Modeling Ingredients*

Ingredients represent the possible options of configurable parameters for repository generation from a domain model. These ingredients are used while defining a recipe. Modeling ingredients as separate concepts and modeling ingredients as attributes of recipes are two complementary modeling paradigms that have been considered for this work. In the effort of finding a balance between the two paradigms, they are compared to one another according to selected criteria: scalability, simplicity, localization of change, and reusability, as shown in Table 6.

Table 6: Modeling ingredients as separate concepts and Modeling ingredients as attributes of recipes.

No.	**Criteria**	**Ingredients as separate concepts**	**Ingredients as attributes of recipes**
1	Scalability	+	-
2	Simplicity	-	+
3	Localization of change	+	-
4	Reusability	+	-

Since the ingredients are expected to grow in the future, the scalability criterion is selected as relevant while modeling ingredients. As can be seen from this comparison table, modeling ingredients as separate concepts is more scalable than modeling ingredients as attributes of recipes with respect to the number of parameters of an ingredient. Modeling ingredients as attributes of recipes is not suitable when the ingredient has its own parameters.

Furthermore, modeling ingredients as separate concepts has scored higher with respect to localization of change. Since the metamodel of recipes need to change in order to support newly introduced ingredients, the criterion localization of change is selected as relevant while modeling ingredients. Over the course of this project, we have observed that the ingredients change more often than the recipes. It is recommended to model concepts that change more often separately from concepts that are stable. Therefore, modeling ingredients as separate concepts has scored higher than modeling ingredients as attributes of recipes with respect to localization of change.

However, modeling ingredients that are not parameterized as separate concepts can be overkill. Modeling ingredients as attributes of recipes is a lightweight alternative when ingredients are not parameterized. For this reason, the modeling ingredients as attributes paradigm has scored higher than modeling ingredients as separate concepts with respect to simplicity. However, in the modeling ingredients as separate concepts paradigm, simplicity is improved by proving a functionality to generate initial ingredients through a wizard.

Modeling ingredients as separate concepts allows the ingredients to be reused over multiple recipes. Modeling ingredients as attributes of recipes makes ingredients tightly coupled with a particular recipe. Reusing these tightly coupled ingredients is not easily possible. Therefore, modeling ingredients as separate concepts has scored higher than modeling ingredients as attributes of recipes with respect to the criterion reusability.

In most of the criteria, since modeling ingredients as a separate concept has scored higher than modeling ingredients as attributes of recipes, the modeling ingredients as separate concepts is used extensively. However, to benefit from the simplicity of modeling ingredients as attributes of recipes, we have decided to apply both approaches whenever appropriate. The DomainModelRecipe and EnumerationRecipes are specifically designed based on the combination of both paradigms.

6.2.1. Modeling Ingredients as separate concepts

The metamodel of the ingredients modeled as separate concepts is shown in Figure 29. The ingredients with a finite number of values that are not expected to grow in the foreseeable future are modeled as Enumerations. Furthermore, the ingredients modeled as Enumerations cannot be parameterized. OrientationKind and CommunicationKind are good examples for this scenario. The ingredients that are expected to be extended are modeled as classes. These ingredients can be extended through parameterization and/or inheritance hierarchy. For this reason, the ingredients Storage and TargetLanguage are modeled as inheritance hierarchies.

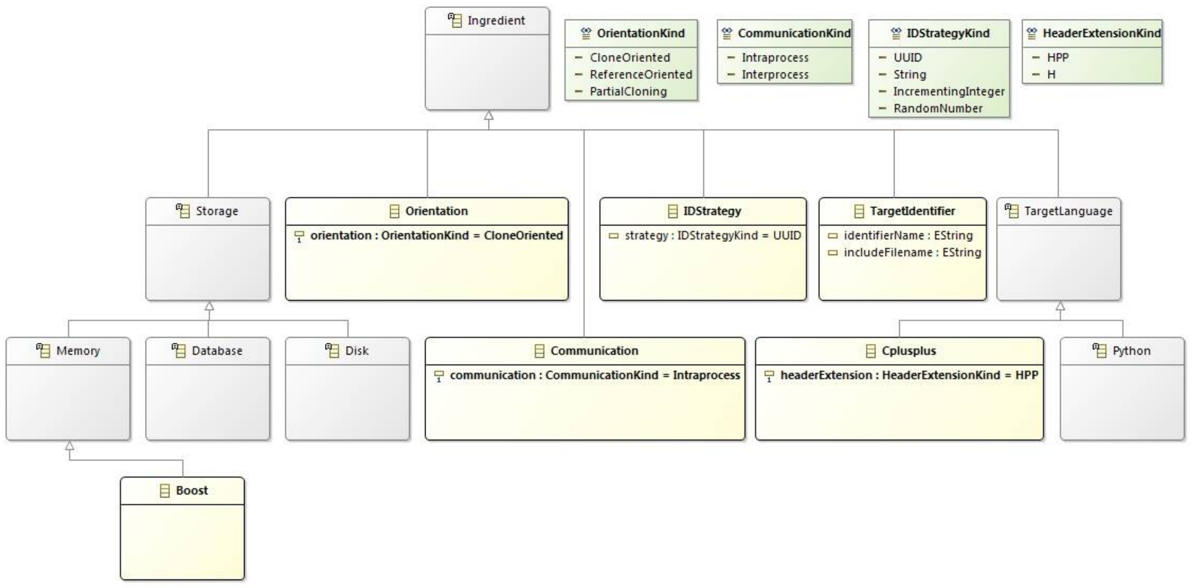


Figure 29: Ingredients Metamodel, Ingredients are modeled as separate concepts

Each of the concepts in the ingredients metamodel is discussed below.

1. Storage

The concept Storage is needed to determine where to store domain models. It provides the options: Memory, Database, and Disk based repositories. The Storage concept is modeled as an abstract super type of the abstract concepts Memory, Database, and Disk. To simplify shared memory management and interprocess communication, the Boost library is chosen as a preferred in memory storage implementation [9]. Boost is the only concrete storage option supported at the moment.

2. Orientation

The concept Orientation is needed to determine how to update repositories containing data as defined by domain models. It provides the options: CloneOriented, ReferenceOriented, and PartialCloning. These three options are modeled as an enumeration, namely OrientationKind.

3. Communication

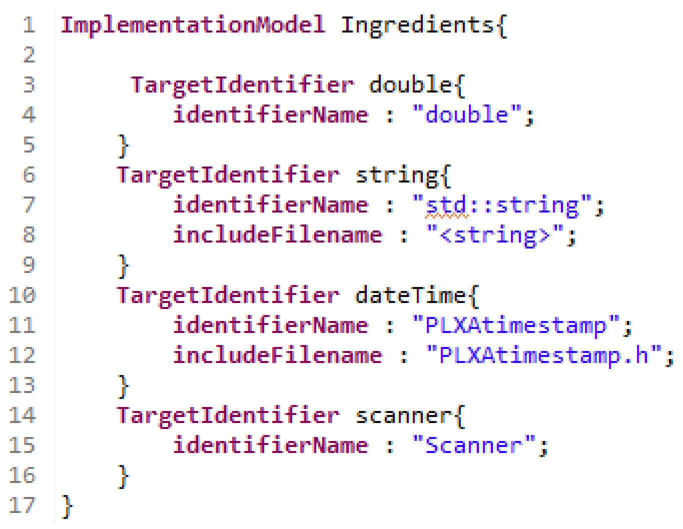
The concept Communication is needed to determine how to share repositories between processes. This concept is modeled as a concrete type of the concept ingredient. Communication provides the options: Intraprocess and Interprocess. These options are modeled as an enumeration, namely CommunicationKind. Intraprocess represents heap (local memory) implementation of repositories that can be used within one process. Interprocess implies a repository implementation that can be shared between processes.

4. ID Strategy

The concept IDStrategy is needed to determine how to uniquely identify Entities in a repository. This concept is modeled as a concrete type of ingredient. IDStrategy provides the options: UUID, String, Incrementing Integer, and Random Number. These options are modeled as an enumeration, namely IDStrategyKind.

5. TargetIdentifier

TargetIdentifier is needed to encapsulate identifier names and, if necessary, include filenames for system and/or user defined files. These TargetIdentifiers are used to map Types in the domain model to corresponding target artifacts during code generation. For example, consider the implementation model snippet shown in Figure 30.



```
ImplementationModel Ingredients{

     TargetIdentifier double{
        identifierName : "double";
    }
    TargetIdentifier string{
        identifierName : "std::string";
        includeFilename : "<string>";
    }
    TargetIdentifier dateTime{
        identifierName : "PLXAtimestamp";
        includeFilename : "PLXAtimestamp.h";
    }
    TargetIdentifier scanner{
        identifierName : "Scanner";
    }
}
```

Figure 30: Example TargetIdentifiers

Each of the TargetIdentifiers in the implementation model snippet above is briefly discussed below.

i. double
This defines an identifier named double. This TargetIdentifier can be used while mapping a Double Type in the domain model to the primitive type double in C++.

ii. string
This defines an identifier named string. This TargetIdentifier can be used while mapping a String Type in the domain model to the primitive type string in C++. This ingredient includes a system file reference for the C++ string primitive type.

iii. dateTime
This defines an identifier named dateTime. This TargetIdentifier is used while mapping dataTime Type in the domain model to a C++ legacy date time. This ingredient includes a user defined file reference for the PLXAtimestamp type.

iv. scanner
This defines an identifier named Scanner. This TargetIdentifier can be used as a target name of the artifacts that must be generated from a Type in the domain model during code generation.

6. TargetLanguage
The concept TargetLanguage is needed to determine which implementation language to use for the repository implementation. This concept is modeled as an abstract subtype of the abstract concept ingredient. TargetLanguage provides the options: C++ and Python. At the moment only C++ is supported with a possibility to choose file extensions for header files. The file extensions for C++ header files are: HPP and H. These file extensions are modeled as enumerations, namely HeaderExtensionKind.

6.2.2. Modeling ingredients as attributes of recipes

In the previous section, the ingredients that are modeled as separate concepts are presented. In this section, the ingredients that are modeled as attributes of recipes are introduced. These ingredients are: component, visibility, and target path. These ingredients are modeled as attributes of the DomainModelRecipe.

1. Component
To minimize configuration complexity, models and the generated code are stored together within one ASML software component. Users are provided with a flexible way of specifying an ASML software component for storing the generated code. This is realized by using the component ingredients, which is modeled as an attribute of the DomainModelRecipe. It gives the option of providing a component to the entire domain model.

2. Visibility
The concept visibility provides a flexible way of specifying whether a model and the corresponding generated code is visible outside of the ASML software component or not. It provides the options: Internal and External. These options are modeled as an enumeration, namely VisibilityKind. Internal models and code are meant to be used within one component. External models and code contain elements that are shared or referred to by other models and code from external ASML software components.

3. TargetPath
The concept targetPath provides a flexible choice of where to store the generated artifacts with respect to the location of the domain model. This is modeled as a string attribute of the DomainModelRecipe.

6.3 Modeling Recipes

Based on an extensive domain analysis and discussions with stakeholders, five kinds of recipes are identified: EntityRecipe, ValueObjectRecipe, TypeRecipe, EnumerationRecipe, and DomainModelRecipe. The Metamodel of these recipes is shown in Figure 31. The getElements() API is used to retrieve the domain model elements that are referred to by each of the recipes.

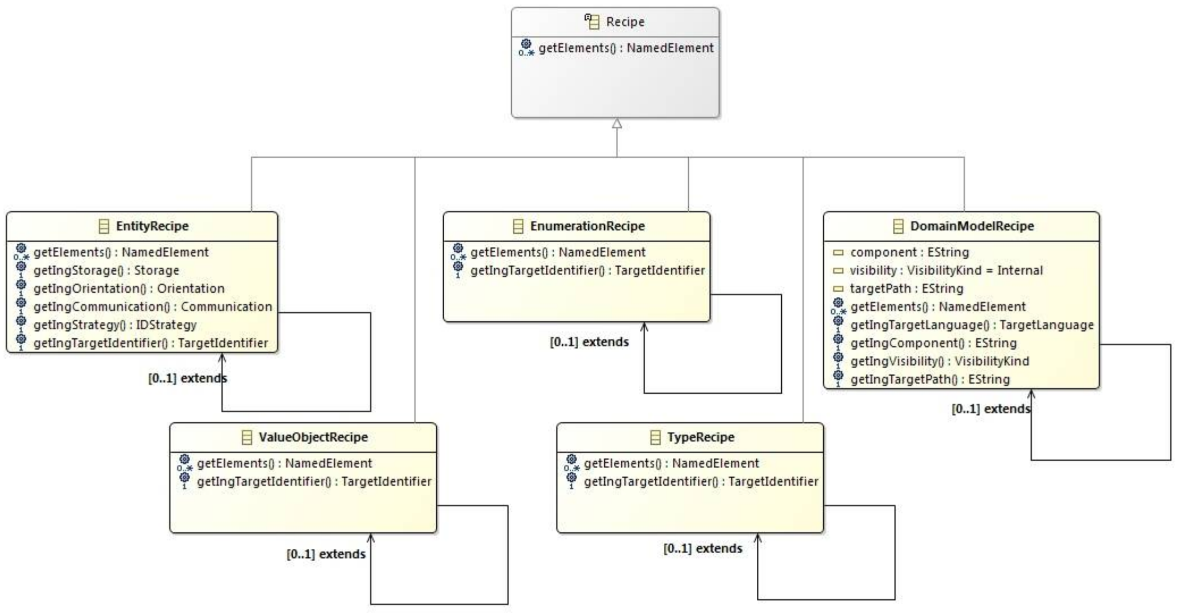

Recipe
getElements() : NamedElement
EntityRecipe
getElements() : NamedElement
getIngStorage() : Storage
getIngOrientation() : Orientation
getIngCommunication() : Communication
getIngStrategy() : IDStrategy
getIngTargetIdentifier() : TargetIdentifier
[0..1] extends
EnumerationRecipe
getElements() : NamedElement
getIngTargetIdentifier() : TargetIdentifier
[0..1] extends
DomainModelRecipe
component : EString
visibility : VisibilityKind = Internal
targetPath : EString
getElements() : NamedElement
getIngTargetLanguage() : TargetLanguage
getIngComponent() : EString
getIngVisibility() : VisibilityKind
getIngTargetPath() : EString
[0..1] extends
ValueObjectRecipe
getElements() : NamedElement
getIngTargetIdentifier() : TargetIdentifier
[0..1] extends
TypeRecipe
getElements() : NamedElement
getIngTargetIdentifier() : TargetIdentifier
[0..1] extends

Figure 31: Recipes Metamodel

To minimize the effort required to create recipes, two design options are considered. These design options allow reusing existing super recipes as much as possible. The first design option is combining the composite pattern with the recipe-ingredient approach to allow nesting sub-recipes within super recipes. The second design option is introducing extension to the recipe-ingredient approach to allow extending super recipes by special ones.

1. Composite pattern
The composite pattern [10] based approach allows nesting specialized sub-recipes inside super recipes. The metamodel of this design approach applied to EntityRecipes and EnumerationRecipes is shown in Figure 32.

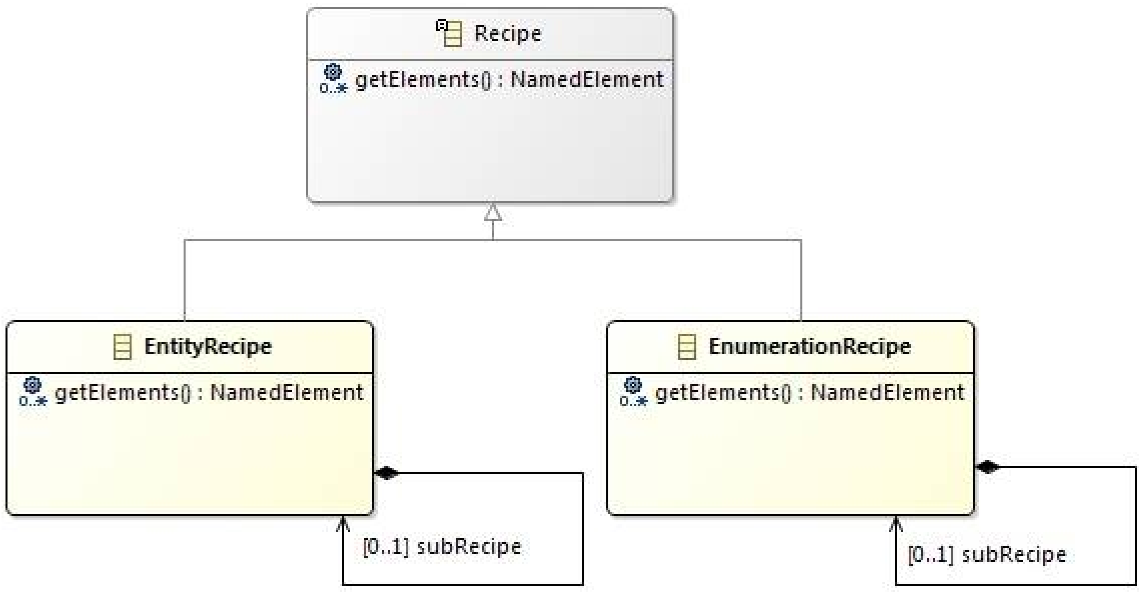
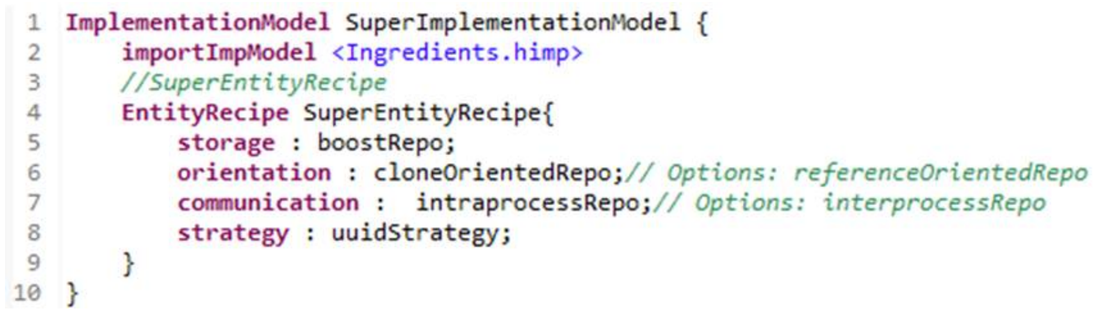

Recipe
getElements() : NamedElement
EntityRecipe
getElements() : NamedElement
EnumerationRecipe
getElements() : NamedElement
[0..1] subRecipe
[0..1] subRecipe

Figure 32: Composite Pattern

This design approach minimizes the effort required to create a recipe by allowing reusing super recipes while defining sub-recipes. For example, the SuperEntityRecipe shown in Figure 33 is reused while defining the specialized CoreModelEntityRecipe, as shown in Figure 34. The SuperEntityRecipe specifies the ingredients: storage, orientation, communication, and strategy.

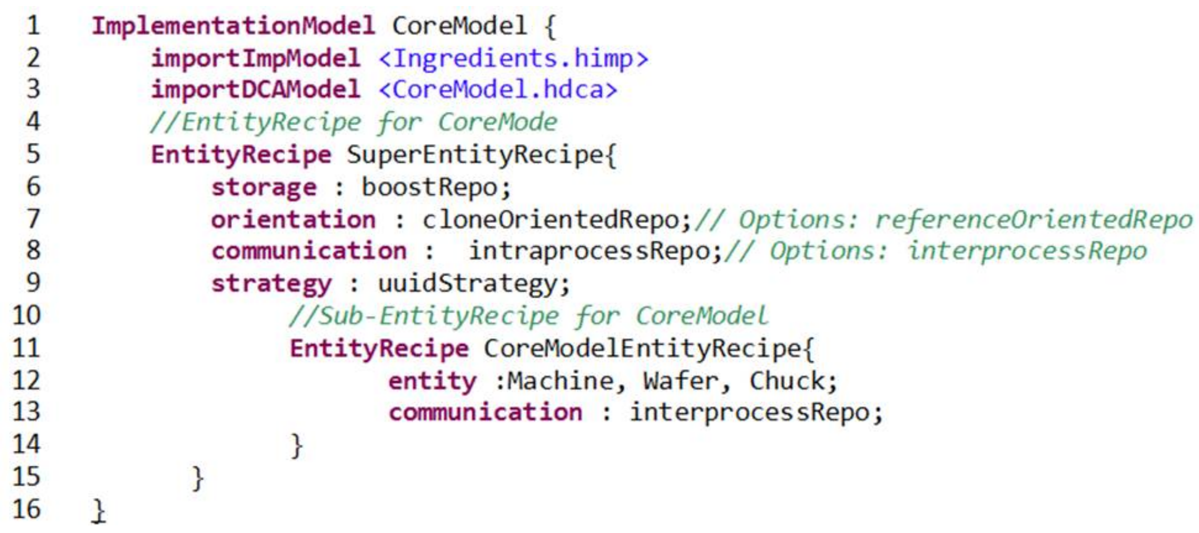


```
ImplementationModel SuperImplementationModel {
    importImpModel <Ingredients.himp>
    //SuperEntityRecipe
    EntityRecipe SuperEntityRecipe{
        storage : boostRepo;
        orientation : cloneOrientedRepo;// Options: referenceOrientedRepo
        communication :  intraprocessRepo;// Options: interprocessRepo
        strategy : uuidStrategy;
    }
}
```

Figure 33: Super EntityRecipe that can be reused in specialized recipes.

Sub-recipes inherit ingredients from their container super recipes. Users can also redefine ingredients in sub-recipes according to their needs. For example, the CoreModelEntityRecipe inherits the ingredients specified by the SuperEntityRecipe and redefines its own communication.

We can see that the CoreModelEntityRecipe is contained inside the SuperEntityRecipe. Therefore, defining specialized sub-recipes require modification of stable super recipes. Since sub-recipes are physically contained inside super recipes, maintaining and creating new sub-recipes require modifying the stable super recipes. This is against the Open/Closed principle of software design. In our context, this principle means that models must be open for extension and closed for modification.



```
ImplementationModel CoreModel {
    importImpModel <Ingredients.himp>
    importDCAModel <CoreModel.hdca>
    //EntityRecipe for CoreMode
    EntityRecipe SuperEntityRecipe{
        storage : boostRepo;
        orientation : cloneOrientedRepo;// Options: referenceOrientedRepo
        communication :  intraprocessRepo;// Options: interprocessRepo
        strategy : uuidStrategy;
            //Sub-EntityRecipe for CoreModel
            EntityRecipe CoreModelEntityRecipe{
                entity :Machine, Wafer, Chuck;
                communication : interprocessRepo;
            }
    }
}
```

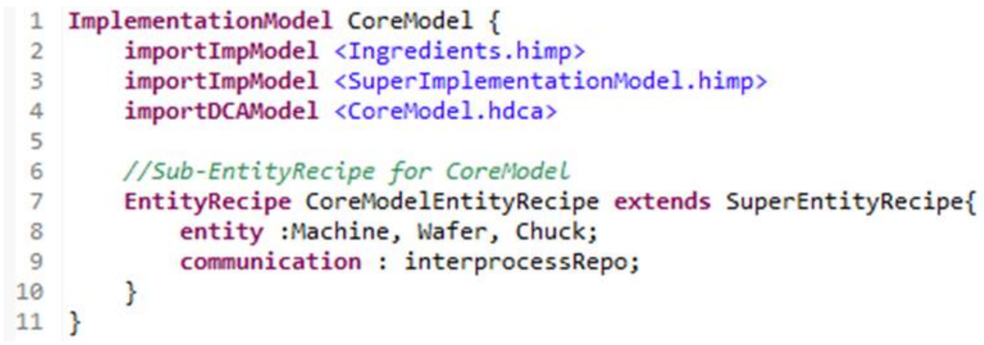
Figure 34: Specialized CoreModelEntityRecipe contained inside SuperEntityRecipe

Furthermore, the composite pattern based approach has a consequence on how ownership is managed within ASML. We have stable implementation models that correspond to core domain models. These implementation models are owned and can only be modified by ASML data architects. ASML software engineers are not allowed to modify these stable implementation models. Software engineers can only reuse these implementation models without modifying them. Unfortunately, in the composite pattern approach, the only way to reuse these stable super recipes is to define specialized sub-recipes inside them, as shown in Figure 34.

2. Extension

In this design approach, specialized recipes extend super recipes. The ingredients in the super recipe are inherited by the specialized ones. Users can also redefine ingredients in the specialized recipes according to their needs. The Entities, ValueObjects, Types, Enumerations, and DomainModels that are referred to by super recipes are not inherited. Every recipe for code generation needs to define its own Entities, ValueObjects, Types, Enumerations, and DomainModels. If it would be possible to inherit Entities, ValueObjects, Types, Enumerations, and DomainModels, the code generator would, for example, generate more than one repository for a single entity. To minimize complexity and ambiguity, special recipes can extend only one super recipe. In this way, we tackled the well-known multiple inheritance diamond problem. In our context, extending multiple recipes would lead to an ambiguity on which version of ingredient to obtain from super recipes. These rules also apply to the composite pattern based approach.

Unlike the composite pattern based approach, this design approach allows reusing stable super recipes without having to modify them. This is realized by the *‘extends’* feature of recipes. Super recipes are physically located outside of the specialized recipes. Therefore, specialized recipes can be maintained without having to modify the stable super recipes and vice versa. For example, the CoreModelEntityRecipe in the implementation model snippet in Figure 35 extends the SuperEntityRecipe in the implementation model snippet in Figure 33.



```
ImplementationModel CoreModel {
    importImpModel <Ingredients.himp>
    importImpModel <SuperImplementationModel.himp>
    importDCAModel <CoreModel.hdca>

    //Sub-EntityRecipe for CoreModel
    EntityRecipe CoreModelEntityRecipe extends SuperEntityRecipe{
        entity :Machine, Wafer, Chuck;
        communication : interprocessRepo;
    }
}
```

Figure 35: Specialized CoreModelEntityRecipe extends SuperEntityRecipe

An ingredient can be defined inside a recipe or inherited from the hierarchy of its super recipes. The order of extension hierarchy determines the value of these ingredients. To find the value of ingredients in a recipe, the ingredient is checked inside this recipe. If the value of the ingredient is not defined inside this recipe, its direct super recipe is checked. If the direct super recipe has defined that ingredient, the search stops and the value of the ingredient is returned. The search for the value of an ingredient continues until the recipe is the highest recipe in the extension hierarchy. In the implementation model snippet in Figure 35, the ingredients: storage, orientation, and ID strategy are not defined inside CoreModelEntityRecipe. Therefore, the direct super recipe of the CoreModelEntityRecipe, which is SuperEntityRecipe is checked. Since the SuperEntityRecipe has defined these ingredients, the search stops and the values are returned. If an ingredient is defined neither in a specialized recipe nor its super recipes, default conventions are applied. To automate this entire search operation, we have developed special getter APIs for each of our ingredients. These special getter APIs are embedded into the definition of the implementation model language in OCLinEcore.

Furthermore, ASML data architects are able to own implementation models for core domain models. This is because ASML software engineers are able to use this core implementation models though the *'extends'* feature without modifying stable super recipes.

The composite pattern and extension based approaches are both implemented and compared to one another with respect to the selected criteria: reusability, maintainability, and ownership. The result of this comparison is shown in Table 7.

Table 7: Comparison of Composite pattern and Extension based design approaches

No.	**Criteria**	**Composite pattern**	**Extension**	**Motivation**
1	Reusability	+	+	As explained in Section 2.5 reusability is one of the design criteria selected in this project. Reusability is selected to minimize the effort required to realize code generation by reusing existing model fragments. Both the composite pattern and extension based approaches allow reusing super recipes.
2	Maintainability	-	+	In the composite pattern based approach, specialized recipes are physically stored inside super recipes, as shown in Figure 34. Therefore, whenever new specialized recipes are introduced or existing ones change, stable super recipes are modified. However, in the extension based approach, specialized recipes are stored outside of super recipes, as shown in the implementation model snippet in Figure 35. Therefore, the extension based approach is more maintainable than the composite pattern based approach.
3	Ownership	-	+	ASML data architects own stable implementation models that correspond to core domain models. These implementation models can be reused by software engineers without modifying them. The composite pattern based approach does not allow reusing these stable implementation models without modifying them. However, the extension based approach allows reusing these stable implementation models without modifying them through the 'extends' feature of recipes. Therefore, the extension based approach is preferred over the composite pattern based approach with respect to the criterion Ownership.

As shown in this comparison table, the extension design option scored higher than the composite pattern based approach. Therefore, the extension design option is selected.

The extension references are introduced across individual recipes instead of the parent super recipe concept. The decision to model the extension references across individual recipes instead of the parent super recipe is to prevent the mistake of extending the wrong recipes. For example, if the extension reference would be in the parent super recipe, as shown in Figure 36, it would be possible to extend EntityRecipes by

EnumerationRecipes. Extending EntityRecipes by EnumerationRecipes is not sensible to our domain. This is because EntityRecipes are intended to only be extended by other specialized EntityRecipes and EnumerationRecipes are intended to only be extended by other specialized EnumerationRecipes.

However, if the extension references are across each of the recipes, as shown in Figure 31, we can only extend recipes that are meaningful according to the rules of our domain. Therefore, the mistake of extending EntityRecipes by EnumerationRecipes is prevented.

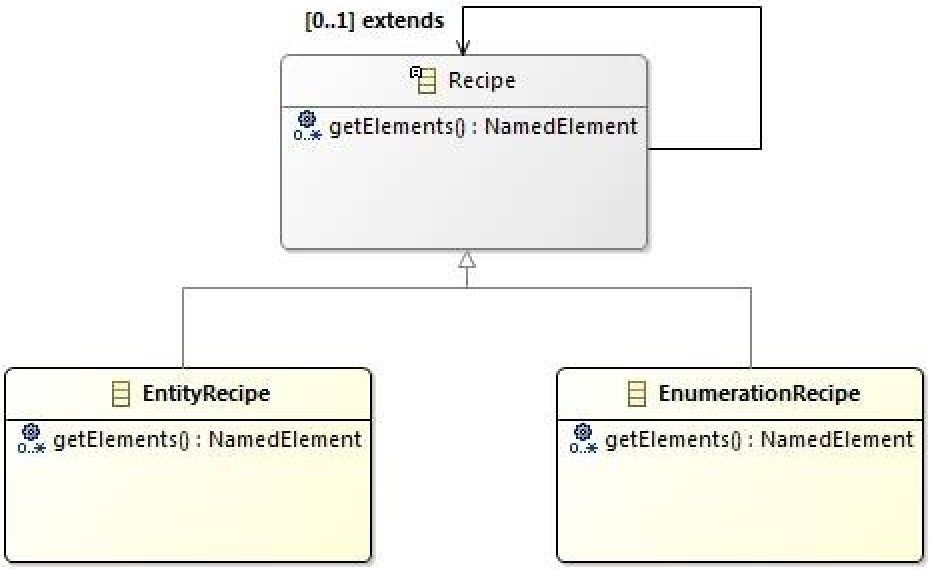
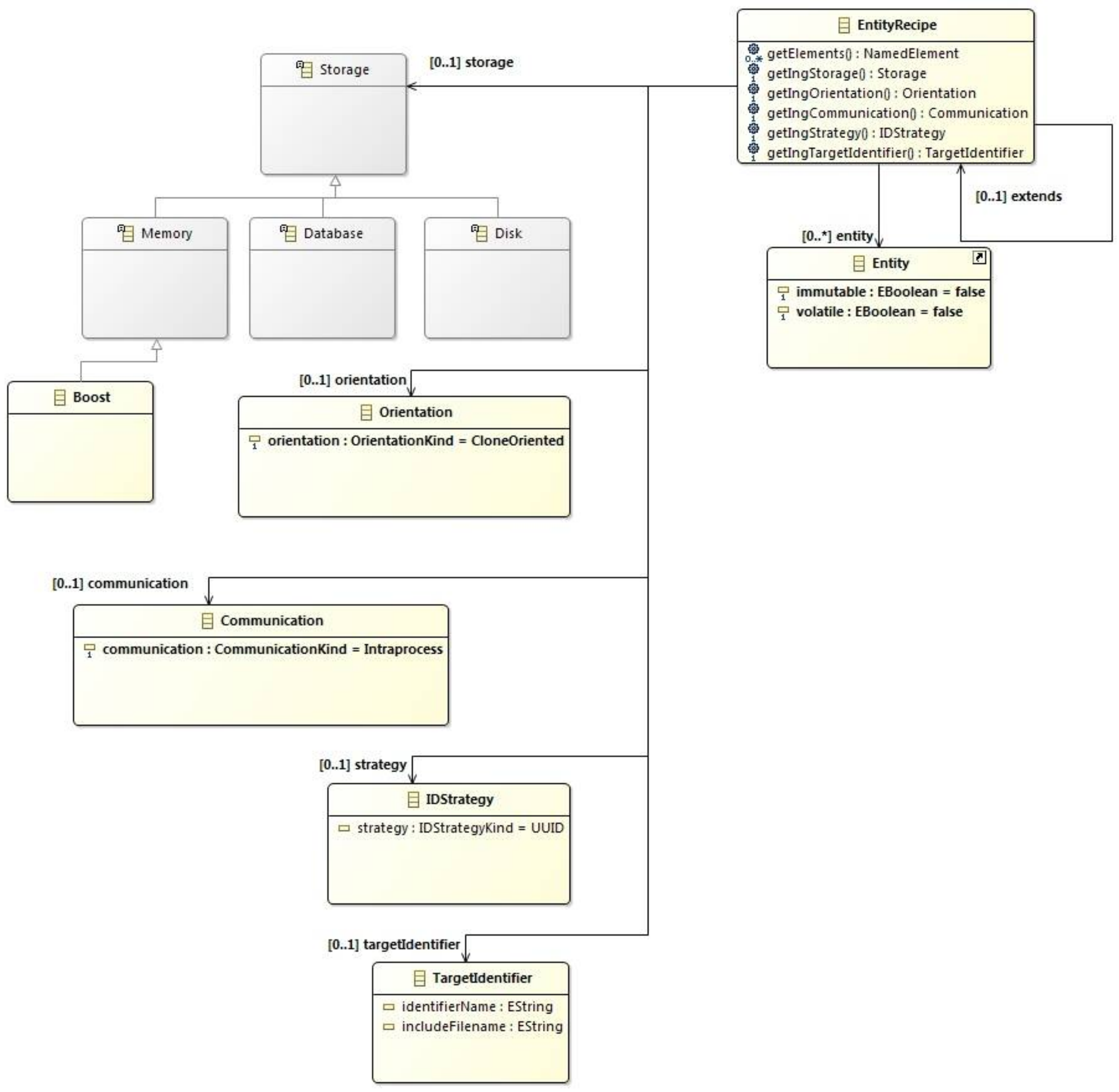

[0..1] extends
Recipe
0..* getElements() : NamedElement
EntityRecipe
0..* getElements() : NamedElement
EnumerationRecipe
0..* getElements() : NamedElement

Figure 36: Extension across the parent recipe concept

6.3.1. EntityRecipe

EntityRecipes with the relevant ingredients are applied to Entities. EntityRecipe contains the choices of ingredients and a reference to the list of Entities whose repositories should be generated based on this recipe. The ingredients that are applicable to EntityRecipe are: Storage, Orientation, Communication, IDStrategy, and TargetIdentifier. An EntityRecipe provides specialized getter APIs for each of these ingredients. The Metamodel of EntityRecipes is shown in Figure 37. Examples of EntityRecipe instances are shown in Figure 33 and Figure 35.

To allow the application of the same EntityRecipe across multiple Entities, the multiplicity of the entity reference relationship is modeled as zero or more. This multiplicity starts from zero to allow creating a super recipe containing the most common ingredients that cannot be applied to any particular entity. Special recipes can extend this super recipe, redefine necessary ingredients, and finally refer to the applicable Entities in the domain model. This applies to all kinds of recipes.



EntityRecipe
getElements() : NamedElement
getIngStorage() : Storage
getIngOrientation() : Orientation
getIngCommunication() : Communication
getIngStrategy() : IDStrategy
getIngTargetIdentifier() : TargetIdentifier
[0..1] storage
Storage
Memory
Database
Disk
Boost
[0..1] extends
[0..*] entity
Entity
immutable : EBoolean = false
volatile : EBoolean = false
[0..1] orientation
Orientation
orientation : OrientationKind = CloneOriented
[0..1] communication
Communication
communication : CommunicationKind = Intraprocess
[0..1] strategy
IDStrategy
strategy : IDStrategyKind = UUID
[0..1] targetIdentifier
TargetIdentifier
identifierName : EString
includeFilename : EString

Figure 37: EntityRecipe Metamodel

6.3.2. ValueObjectRecipe

ValueObjectRecipes with the relevant ingredients are applied to ValueObjects. ValueObjectRecipe contains the choices of ingredients and a reference to the list of ValueObjects whose code should be generated based on this recipe. The ingredient that is applicable to ValueObjectRecipe is TargetIdentifier. A ValueObjectRecipe provides a specialized getter API for the ingredient TargetIdentifier. The Metamodel of ValueObjectRecipes is shown in Figure 38. An example ValueObjectRecipe instance is shown in the implementation model snippet in Figure 39.

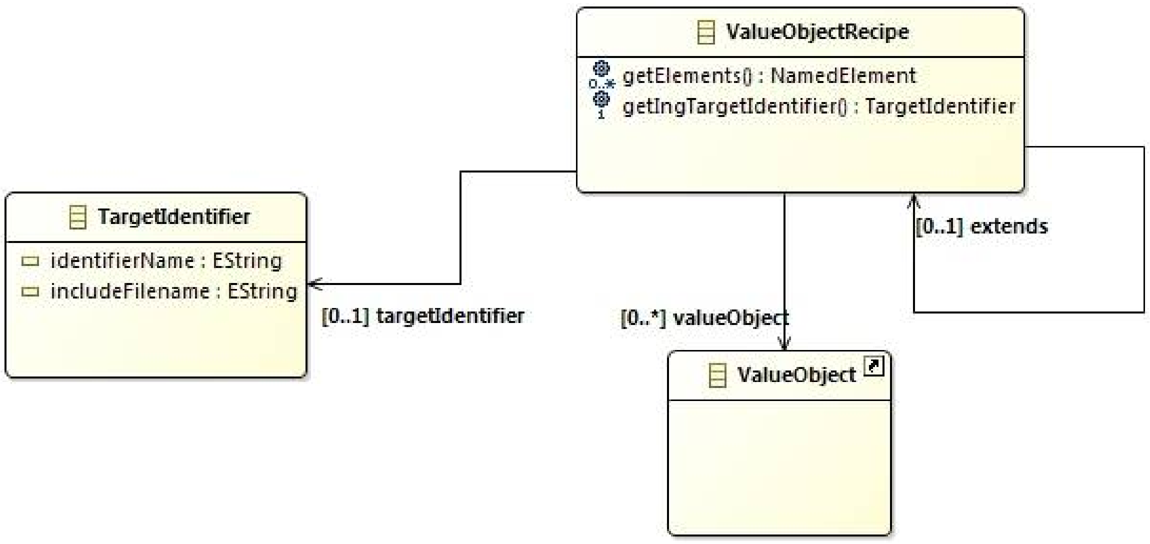
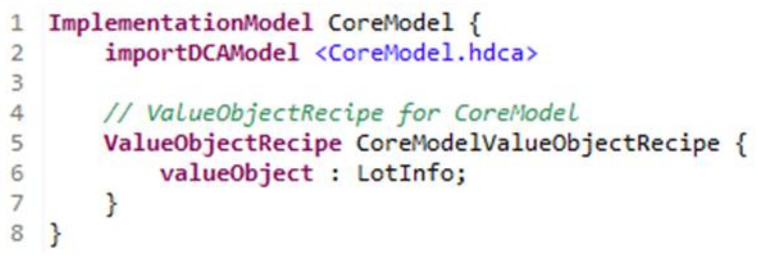


Figure 38: ValueObjectRecipe Metamodel

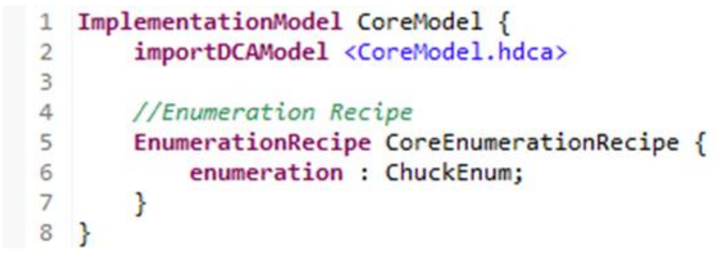


```
ImplementationModel CoreModel {
    importDCAModel <CoreModel.hdca>

    // ValueObjectRecipe for CoreModel
    ValueObjectRecipe CoreModelValueObjectRecipe {
        valueObject : LotInfo;
    }
}
```

Figure 39: CoreModelValueObjectRecipe, ValueObjectRecipe instance

6.3.3. EnumerationRecipe

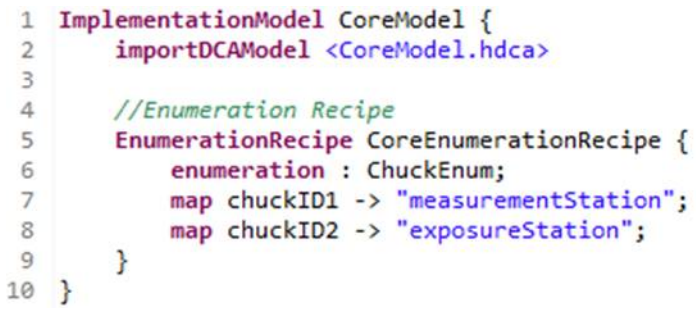
EnumerationRecipes with the relevant ingredients are applied to Enumerations. EnumerationRecipe contains the choices of ingredients and a reference to the list of Enumeration Types whose code should be generated based on this recipe. The ingredient that is applicable to EnumerationRecipe is TargetIdentifier. An EnumerationRecipe provides a specialized getter API for the ingredient TargetIdentifier. The Metamodel of EnumerationRecipes is shown in Figure 42.



```
ImplementationModel CoreModel {
    importDCAModel <CoreModel.hdca>

    //Enumeration Recipe
    EnumerationRecipe CoreEnumerationRecipe {
        enumeration : ChuckEnum;
    }
}
```

Figure 40: CoreEnumerationRecipe, EnumerationRecipe instance without out LiteralMappings

EnumerationRecipe contains zero or more LiteralMappings. The concept LiteralMapping is introduced to EnumerationRecipes to allow mapping EnumerationLiterals in the domain model into enumeration literals in a legacy code.



```
ImplementationModel CoreModel {
    importDCAModel <CoreModel.hdca>

    //Enumeration Recipe
    EnumerationRecipe CoreEnumerationRecipe {
        enumeration : ChuckEnum;
        map chuckID1 -> "measurementStation";
        map chuckID2 -> "exposureStation";
    }
}
```

Figure 41: CoreEnumerationRecipe, EnumerationRecipe instance with EnumerationLiterals

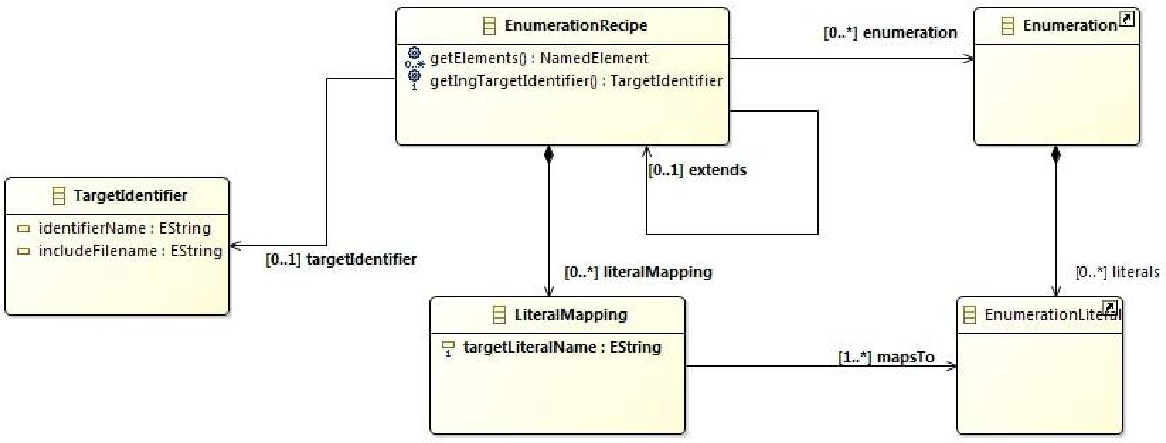
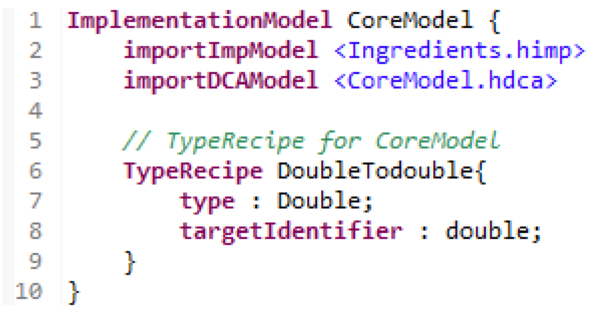


Figure 42: EnumerationRecipe Metamodel

6.3.4. TypeRecipe

TypeRecipes with the relevant ingredients are applied to Types. TypeRecipe contains the choices of ingredients and a reference to the list of Types whose code should be generated based on this recipe. The ingredient that is applicable to TypeRecipe is TargetIdentifier. A TypeRecipe provides a specialized getter API for the ingredient TargetIdentifier. The Metamodel of EnumerationRecipes is shown in Figure 44.



```
ImplementationModel CoreModel {
    importImpModel <Ingredients.himp>
    importDCAModel <CoreModel.hdca>

    // TypeRecipe for CoreModel
    TypeRecipe DoubleTodouble{
        type : Double;
        targetIdentifier : double;
    }
}
```

Figure 43: DoubleTodouble, TypeRecipe instance

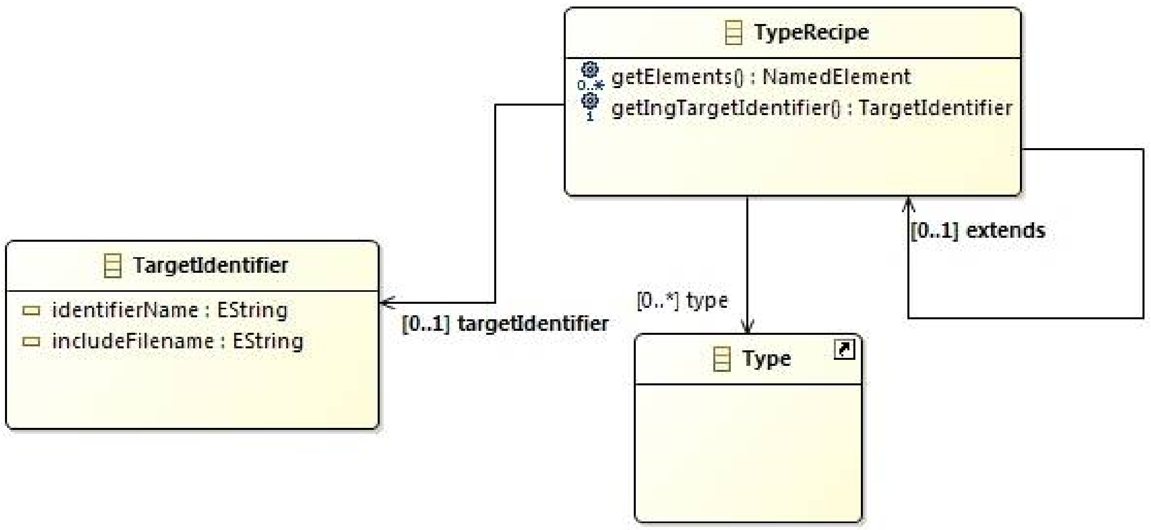

TypeRecipe
0..* getElements() : NamedElement
1 getIngTargetIdentifier() : TargetIdentifier
[0..1] extends
TargetIdentifier
identifierName : EString
includeFilename : EString
[0..1] targetIdentifier
[0..*] type
Type

Figure 44: TypeRecipe Metamodel

6.3.5. DomainModelRecipe

DomainModelRecipes with the relevant ingredients are applied to domain models. DomainModelRecipe contains the choices of ingredients and a reference to the list of domain models whose code should be generated based on this recipe. The ingredients that are applicable to DomainModelRecipe are: component, visibility, targetPath, and TargetLanguage. A DomainModelRecipe provides a specialized getter API for the ingredient TargetLanguage. The Metamodel of DomainModelRecipes is shown in Figure 45.

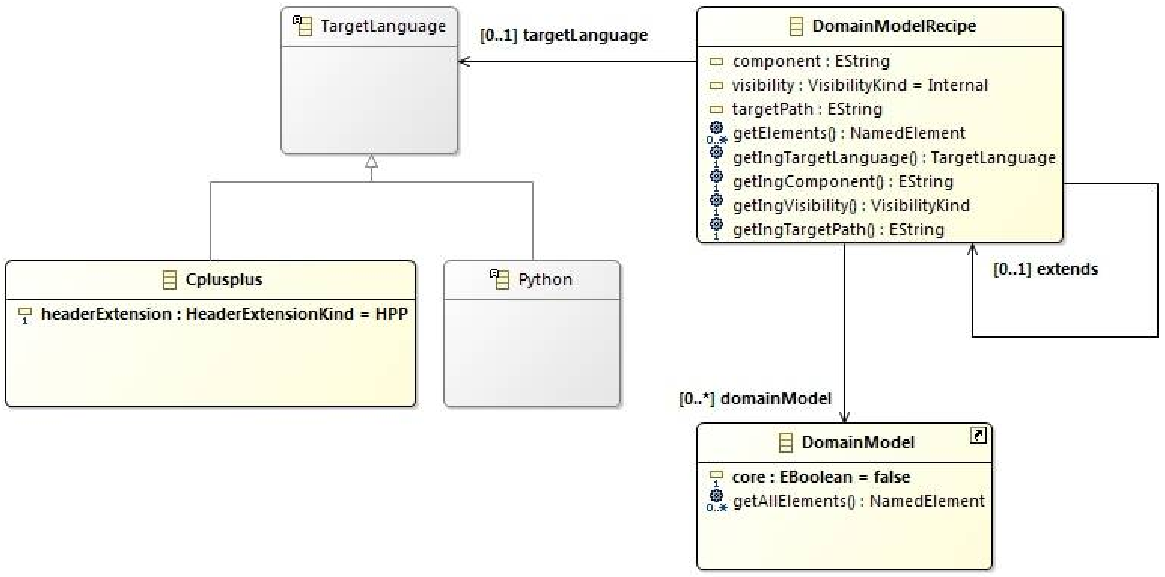
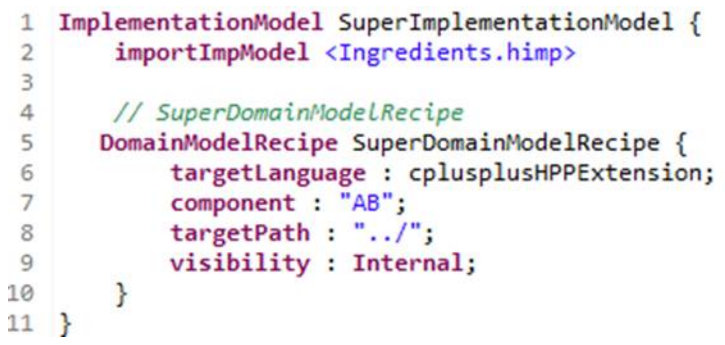
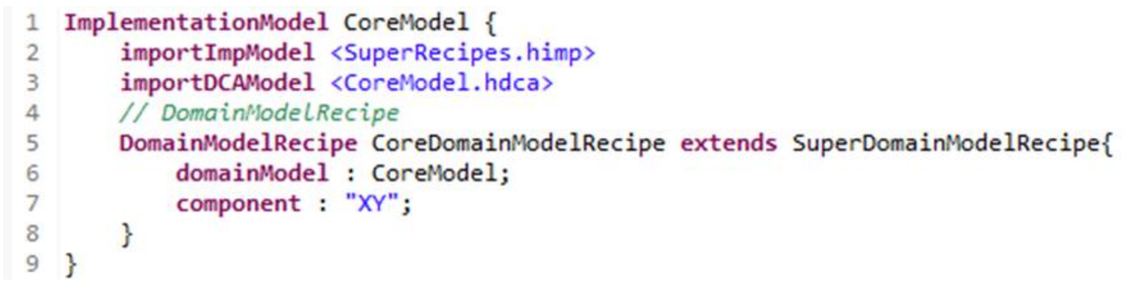

TargetLanguage
[0..1] targetLanguage
DomainModelRecipe
component : EString
visibility : VisibilityKind = Internal
targetPath : EString
0..* getElements() : NamedElement
1 getIngTargetLanguage() : TargetLanguage
1 getIngComponent() : EString
1 getIngVisibility() : VisibilityKind
1 getIngTargetPath() : EString
[0..1] extends
Cplusplus
1 headerExtension : HeaderExtensionKind = HPP
Python
[0..*] domainModel
DomainModel
1 core : EBoolean = false
0..* getAllElements() : NamedElement

Figure 45: DomainModelRecipe Metamodel



```
ImplementationModel SuperImplementationModel {
    importImpModel <Ingredients.himp>

    // SuperDomainModelRecipe
   DomainModelRecipe SuperDomainModelRecipe {
        targetLanguage : cplusplusHPPExtension;
        component : "AB";
        targetPath : "../";
        visibility : Internal;
    }
}
```

Figure 46: SuperDomainModelRecipe, DomainModelRecipe instance



```
ImplementationModel CoreModel {
    importImpModel <SuperRecipes.himp>
    importDCAModel <CoreModel.hdca>
    // DomainModelRecipe
    DomainModelRecipe CoreDomainModelRecipe extends SuperDomainModelRecipe{
        domainModel : CoreModel;
        component : "XY";
    }
}
```

Figure 47: CoreDomainModelRecipe extending SuperDomainModelRecipe

6.4 Implementation Model Editor

For usability reasons, such as speed and convenience, stakeholders preferred a textual editor for the implementation model language over a graphical one. Therefore, only a textual editor is provided. Since EMFText [11] and Xtext [12] can be used for specifying the textual editors of Ecore-based metamodels, they are both considered for this work. To select the most suitable language from these two competitors, they are compared to one another with respect to the criteria: simplicity, expertise, community support, and suitability. The result of this comparison is shown in Table 8.

Table 8: Choice of a Language for Textual Syntax specification

No.	**Criteria**	**EMFText**	**Xtext**	**Motivation**
1	Simplicity	+	-	EMFText is a lightweight syntax specification language compared to Xtext.
2	Expertise	+	-	ASML SW Architects have used EMFText to specify the textual syntax of the domain model language. Therefore, it was possible to find local support whenever necessary.
3	Community support	-	+	Xtext is the standard textual modeling language in the Eclipse community. It has more active developer community support than EMFText.
4	Suitability	+	-	Xtext derives metamodels from the syntax specification. While existing Ecore-based metamodels can be used to generate a textual editor, this is not a natural fit for Xtext. However, EMFText does not derive metamodels from the syntax specification. Existing Ecore-based metamodels are used as a basis for creating textual editors. Since we have designed the implementation model language Ecore-based metamodel before defining the syntax, EMFText is more suitable for our purpose than Xtext.

As can be seen from the comparison table, EMFText scored higher than Xtext in most of our criteria. Therefore, EMFText is selected as the textual syntax specification language for the implementation model language.

6.5 *Template Language Selection*

Various components in the reference architecture, such as the Repository Generator, Implementation Model Wizard, and Implementation Model Validation components are entirely or partially designed based on the M2T approach of the MDA. Therefore, it was necessary to select an appropriate template language for realizing the M2T approach. The Acceleo [13] and Xtend [14] template languages are considered for realizing M2T in this work. To select the most appropriate M2T language for this project, these two template languages are compared to one another with respect to the criteria: expertise, standardization, and suitability, as shown in Table 9.

Table 9: Choice of a Template Language for realizing M2T

No.	**Criteria**	**Acceleo**	**Xtend**	**Motivation**
1	Expertise	+	-	Since we have a better expertise in Acceleo than Xtend, Acceleo is the preferred M2T language than Xtend with respect to expertise.
2	Standardization	+	-	Acceleo is a standard code generation template language based on the OMG specification for code generation.
3	Suitability	+	+	Acceleo and Xtend are both suitable for code generation based on EMF models. Xtend integrates seamlessly with Java. Acceleo provides a full-featured IDE for developing code generation and a back door for accessing Java services.

As can be seen from the comparison table above, the Acceleo template language has scored higher in two of the three criteria. Therefore, the Acceleo template language is selected as the M2T language in this project.

6.6 *Repository Generator Design*

This component is responsible for generating the right repositories based on the ingredients of a recipe. In addition to the ingredients specified in each of the recipes, this component considers the choices made at the domain modeling level. These domain modeling level choices are volatility, immutability, multiplicities, and the relationships between the domain model elements. The repository generator component consists of the various modules shown in the dependency graph in Figure 48.

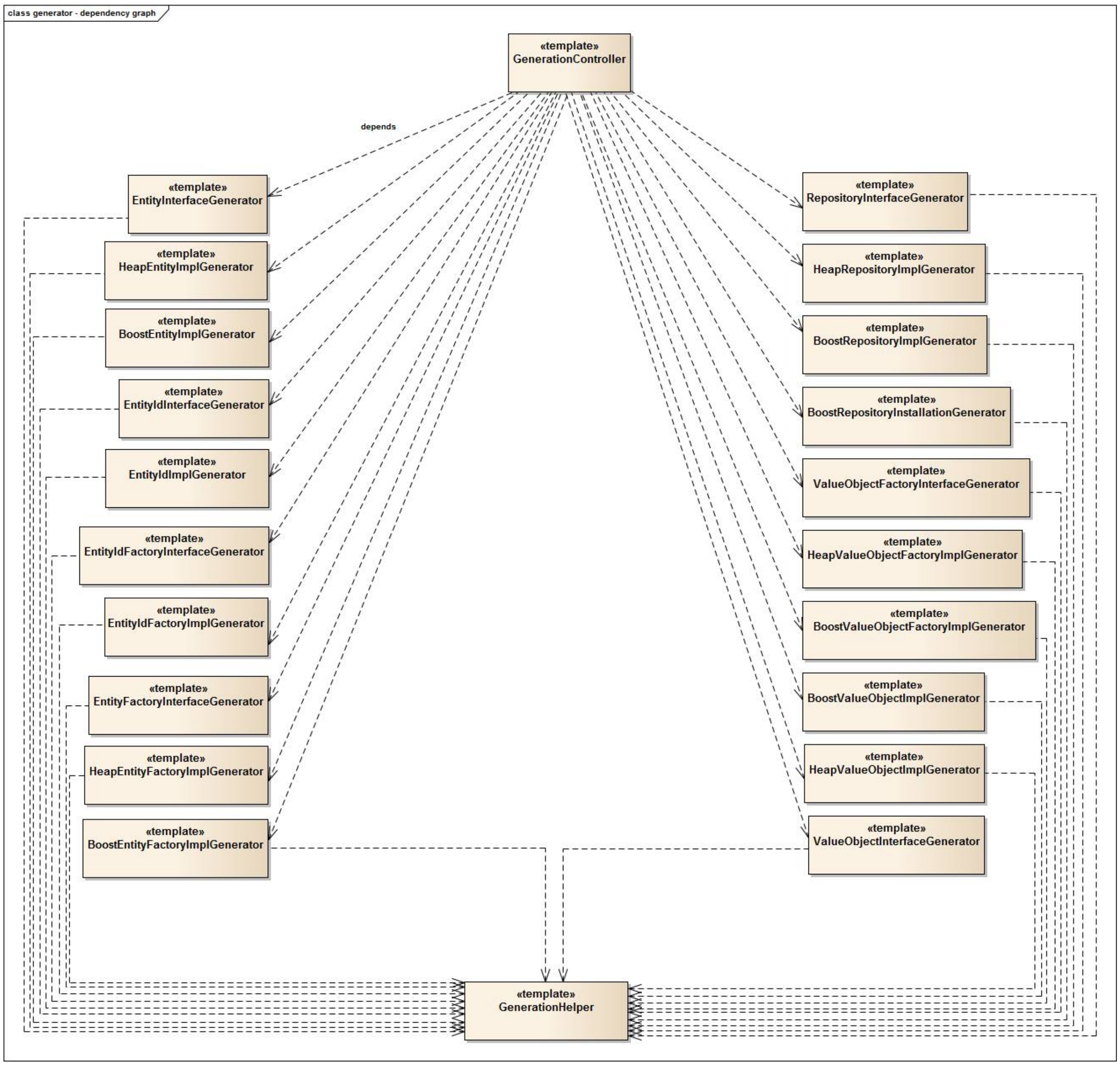

class generator - dependency graph
depends
«template» GenerationController
«template» EntityInterfaceGenerator
«template» HeapEntityImplGenerator
«template» BoostEntityImplGenerator
«template» EntityIdInterfaceGenerator
«template» EntityIdImplGenerator
«template» EntityIdFactoryInterfaceGenerator
«template» EntityIdFactoryImplGenerator
«template» EntityFactoryInterfaceGenerator
«template» HeapEntityFactoryImplGenerator
«template» BoostEntityFactoryImplGenerator
«template» RepositoryInterfaceGenerator
«template» HeapRepositoryImplGenerator
«template» BoostRepositoryImplGenerator
«template» BoostRepositoryInstallationGenerator
«template» ValueObjectFactoryInterfaceGenerator
«template» HeapValueObjectFactoryImplGenerator
«template» BoostValueObjectFactoryImplGenerator
«template» BoostValueObjectImplGenerator
«template» HeapValueObjectImplGenerator
«template» ValueObjectInterfaceGenerator
«template» GenerationHelper

Figure 48: Dependency graph, showing the dependency between the template modules that realize the repository generator.

The various modules in the RepositoryGenerator component can be grouped into three categories based on responsibility. These are the GenerationController, the various generation modules, and the GenerationHelper. Each of these categories is discussed below.

6.6.1. GenerationController

This module is the central controller of all the other transformation modules. It determines which transformation module gets called based on the ingredients in a recipe. The main challenges in the design of this module are: *how to extract the ingredients from a recipe and how to better organize the other modules to generate code from the domain model elements that are referred to from this recipe*. To tackle this challenge, two design approaches are considered. These approaches are: Visitor/Decorator and Straightforward approaches. Each of these two design options are briefly discussed below.

1. Visitor/Decorator based approach

This approach is based on the visitor and decorator design patterns [10]. Typically, the strategy that comes to mind when confronted with the problem of visiting an object structure is the visitor pattern. The visitor pattern is used to visit every recipe and extract the corresponding ingredients and the referred domain model elements. Furthermore, the visitor pattern is used to visit domain model structures to generate code for each of the elements that are referred to from this recipe.

Since neither the domain model nor the implementation model languages are designed having the visitor pattern in mind, directly applying the visitor pattern to the implementation and domain models is not possible. The difficulty is that these models do not have the *accept(Visitor :Object)* method. This problem is tackled with the decorator pattern. The *Recipes* of the implementation model language and *Types* of the domain model language are decorated with the *accept(Visitor :Object)* method.

2. Straightforward approach

In this approach, the extraction of the domain model elements from a recipe is performed sequentially. The controller iterates over the implementation model, taking all the recipes into account. The extraction of Entities and ValueObjects to generate Entity interfaces, repository interfaces, and ValueObject interfaces is shown in the sequence diagram in Figure 49.

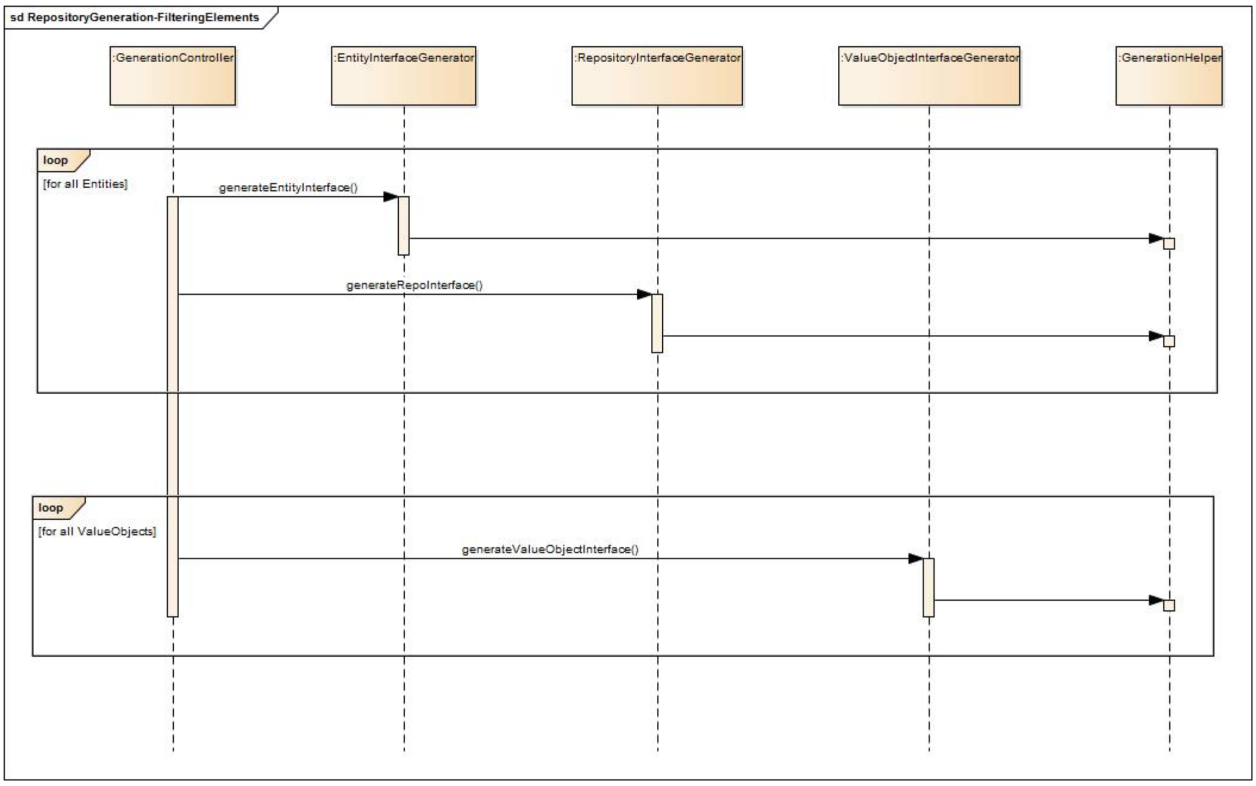

sd RepositoryGeneration-FilteringElements
:GenerationController
:EntityInterfaceGenerator
:RepositoryInterfaceGenerator
:ValueObjectInterfaceGenerator
:GenerationHelper
loop
[for all Entities]
generateEntityInterface()
generateRepoInterface()
loop
[for all ValueObjects]
generateValueObjectInterface()

Figure 49: Iteration over the implementation model to extract Entities and ValueObjects for code generation

The controller determines the ingredients needed for a specific generation task through sequence of decisions until the remaining decisions are trivial enough to be handled by the individual generation templates. The decisions taken to determine the ingredients for code generation from Entities are shown in the activity diagram in Figure 50.

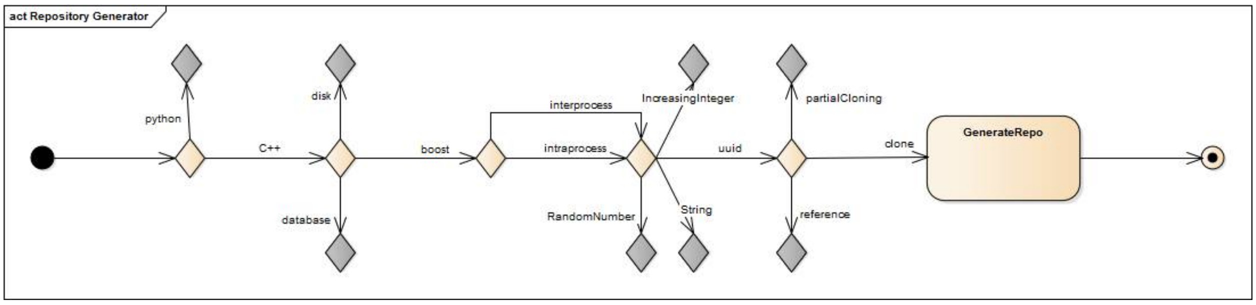


Figure 50: Sequence of decisions made to determine the ingredients needed for a specific code generation task

To select the most appropriate approach out of these two candidates, they are compared to one another based on the criteria: design reuse, fitness for selected technology, code analyzability, ease of use from controller, lines of code (LOC), and localization of change. The result of this comparison is shown in Table 10.

Table 10: Comparison of code generation approaches

No.	**Criteria**	**Visitor/Decorator**	**Straightforward**	**Motivation**
1	Design reuse	+	-	As indicated by its name, the Visitor/Decorator based approach is based on the visitor and decorator design patterns. The design and implementation strategies of these patterns can safely be reused and tailored to solve the problems at hand. However, the Straightforward approach is developed from scratch.
2	Fitness for selected Technology	-	+	Acceleo is selected as the model-to-text transformation template language in this project. Unfortunately, the visitor and decorator patterns are not a natural fit for this selected template language. As indicated by the name, the Straightforward approach was straightforward to implement in Acceleo.
3	Code Analyzability	-	+	To implement the Visitor/Decorator patterns in Acceleo, a number of unnecessary indirections are introduced. This makes the code less analyzable compared to its counterpart implementation in the straightforward approach.
4	Ease of use from controller	+	-	In the Visitor/Decorator based approach, users of the GenerationController need to call only two methods: *visit(Visitor: Object)* and *accept(Element: Object).* In the Straightforward approach, many methods are involved.
5	Lines of code (LOC)	-	+	For comparison purposes the same functionality of the BoostIntraprocess repositories are implemented in both design approaches. The Visitor/Decorator based approach was implemented in 129 lines of code and the Straightforward approach in 111.
6	Localization of Change	+	+	For comparison purposes a new hypothetical recipe is introduced into the domain model language. Localization of the changes required to support this new recipe were comparable in both design approaches.

According to the result of this comparison, the Straightforward approach has scored higher than the Visitor/Decorator based approach. Therefore, the Straightforward approach is selected as the design strategy for realizing the Repository Generator.

6.6.2. Generators

The various generators for repositories, Entities, ValueObjects, and Factories are responsible for generating the right code based on the ingredients in their corresponding recipes. These generators are coordinated by the GenerationController module.

6.6.3. GenerationHelper

The GenerationHelper module contains a number of queries that assists the various generators during code generation. This GenerationHelper does not depend on other modules.

6.6.4. Supported features

Repository generation is supported for the features summarized in Table 11.

Table 11: Summary of features supported by the code generator.

Category	Features	Supported Options
Implementation Model /Ingredients	Storage	Boost
	Orientation	Clone
	Communication	Intraprocess / Heap
		Interprocess / Shared Memory
	IDStrategy	UUID
	TargetIdentifier	
	TargetLanguage	C++, H and HPP header extensions
	Component	
	Visibility	Internal
		External
	targetPath	
Domain Model	Entity	Volatile
		Mutable
		Contains ValueObject
		Contains Enumeration
		Contains Type
		Contains legacy type
		Association with other Entity
		Elements can be from other models, the same namespaces.
	EntityMultiplicity	Zero or more
	ValueObject	Contains ValueObject
		Contain Enumeration
		Contains Type
		Association with other Entity
		Elements can be from other models, the same namespaces.
	Enumerations	
	Multiplicities	Zero or more
		Integer
		Infinity
		MultiplicityConstant

6.7 *Implementation Model Wizard*

The repository generation tool provides a wizard for generating default implementation models from both the graphical and textual representations of domain models. This wizard helps users in learning and getting quickly started with the tool. Users can use the wizard to easily generate initial implementation models containing all kinds of recipes and ingredients. These implementation models can be used for learning the implementation model language and its syntax. Furthermore, these models can also be reused in other implementation models as much as possible. For example, the ingredients can be reused while defining other recipes.

Two design approaches are considered for realizing this implementation model wizard: model-to-model transformation (M2M) and model-to-text transformation (M2T). These two design approaches are compared to one another based on selected relevant criteria: expertise and flexibility.

The design and implementation of the Repository Generator component is based on the M2T approach. During the design and implementation of this component, we developed expertise in the design techniques, tools, and languages of the M2T based approach. Therefore, the implementation model wizard can be realized in a short period of time by applying the M2T approach. The downside of this approach is that the wizard needs to change whenever the concrete syntax of the implementation model language changes. This makes the wizard less flexible. The flexibility could have been improved following the M2M based approach. In the M2M approach, models can be created independent of the concrete syntax. These models can later be serialized using the concrete syntax. However, extra time and effort is required to gain the same level of expertise as in the M2T based approach. Therefore, given the short time budget, we decided to realize the implementation model wizard by applying our accumulated expertise in the M2T based design approach.

6.7.1. Default Conventions

The Implementation Model Wizard creates initial implementation models by applying default conventions. After comprehensive analysis of the domain and discussions with stakeholders, the relevant default conventions for the initial implementation model created through the wizard are identified. These identified default conventions and the motivations behind these choices are show in Table 12.

Table 12: Implementation model default conventions for the wizard.

Implementation Model Element	Ingredient/ applicable to	Default	Motivation
EntityRecipe	Storage	Boost	This is the only supported type of storage implementation strategy.
	Orientation	Clone	This is the only supported orientation at the moment.
	Communication	Intraprocess	Simplicity
	ID Strategy	UUID	This is the only supported ID Strategy at the moment.
	Target Identifier	Entity name	Simplicity
	Entities that this recipe is applicable	All Entities in the domain model	Convention
DomainModelRecipe	Component	No default	convention
	Target path	The same location as domain model	convention
	Target Language	C++, HPP	C++ is a requirement, HPP is a convention.
	Domain Models that this recipe is applicable	The domain model corresponding to the implementation model.	Convention
ValueObjectRecipe	Target Identifier	ValueObject name	Simplicity
	ValueObjects that this recipe is applicable	All ValueObjects	convention
EnumerationRecipe	Target Identifier	Enumeration name	Simplicity
	Enumeration that this recipe is applicable	All enumerations	Convention
	Literal Mapping	Mapped to exactly the same literals	Convention
TypeRecipe	Target Identifier	Type name	Simplicity
	Types that this recipe is applicable	All primitive types	Convention

6.8 Implementation Model Validation

The Implementation Model Validation component is responsible for validating the input Implementation Models for code generation. An early detection of incorrect models prevents the problem of generating code that does not build, or worse, a code that builds but produces incorrect results when deployed in the TWINSCAN system. To tackle this problem, various validation rules are identified. The main validation rules are given below.

1. All elements for code generation must have a recipe.
2. All elements for code generation must have exactly one recipe.
3. All ingredients of the recipes for code generation must be explicitly set i.e. no default values are allowed during code generation except TargetIdentifiers.
4. Visibility in the implementation models
    a. External can refer to any other External.
    b. External can never refer to any Internal.
    c. Internal can refer to Internal only within the same component.
    d. Internal can refer to External within any component.
5. There are 3 classes of implementation models (for a style purpose combination is not allowed)
    a. An implementation model must contain only ingredients.
    b. An implementation model must contain only super recipes.
    c. An implementation model must contain code generation recipes for exactly one domain model and the elements within it.
6. All elements in the domain model must have recipes in the implementation model (style).
7. Unsupported features must not be selected in the implementation model.

Two design options are considered during the realization of the Implementation Model Validation: OCL invariants and Acceleo template and/or query. To select the most appropriate design approach, the two are compared to one another with respect to maturity. The maturity criterion refers to how well these two approaches integrate with the Eclipse Modeling Framework. The Eclipse Modeling Framework provides a runtime environment to the repository generation tool, as shown in Figure 25. The Acceleo template and/or query are more matured than the OCL invariants with respect to integration with the Eclipse Modeling Framework. Due to the immaturity of the environment, the OCL invariants-based approach presented error messages that cannot be easily related to the cause of these errors. The Acceleo template and/or query -based approach produces readable custom error messages defined by ourselves. Therefore, the Acceleo template and/or query -based approach is selected as the most appropriate design approach for realizing the Implementation Model Validation component.

The Implementation Model Validation is designed to work in two complementary ways. These are:

1. Validation Without code generation

The goal of this Implementation Model Validation is to be able to identify errors without code generation. This could be before and/or after code generation. Validation without code generation is described in the sequence diagram shown in Figure 51.

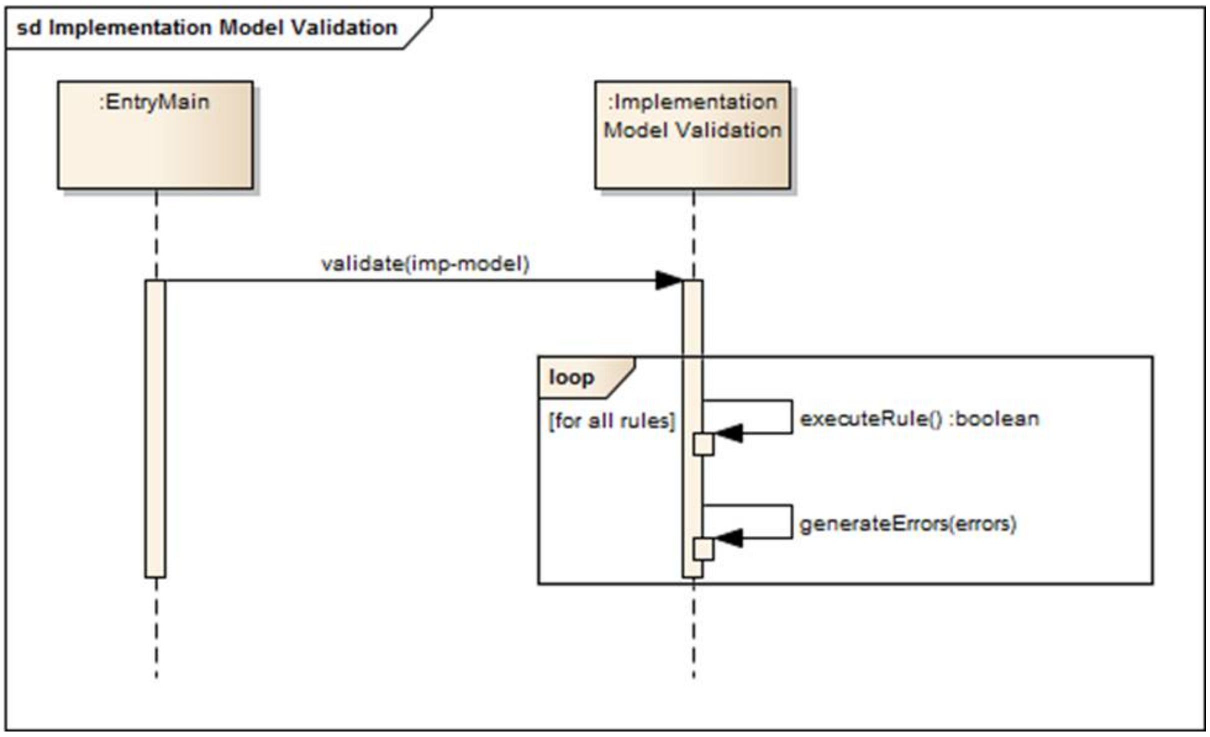


Figure 51: Validation without code generation

2. Validation during code generation

The goal of this Implementation Model Validation is to identify errors during code generation. Validation during code generation is described in the sequence diagram shown in Figure 52.

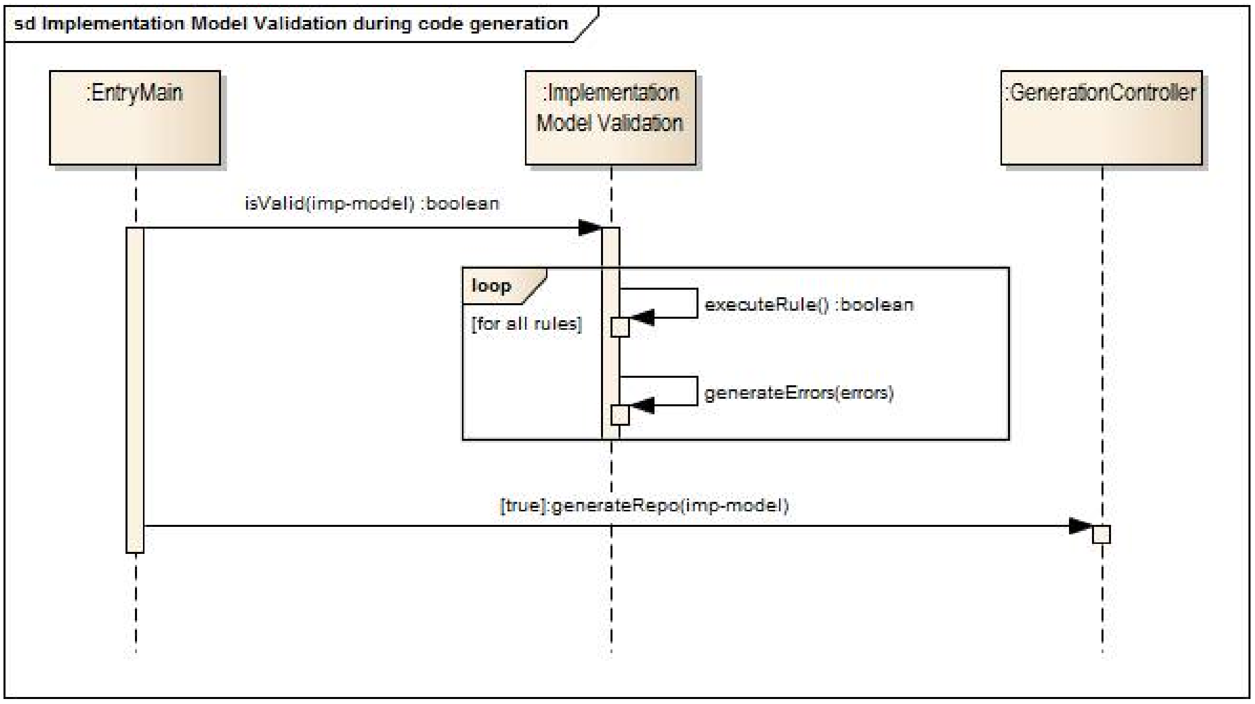

sd Implementation Model Validation during code generation
:EntryMain
:Implementation Model Validation
:GenerationController
isValid(imp-model) :boolean
loop
[for all rules]
executeRule() :boolean
generateErrors(errors)
[true]:generateRepo(imp-model)

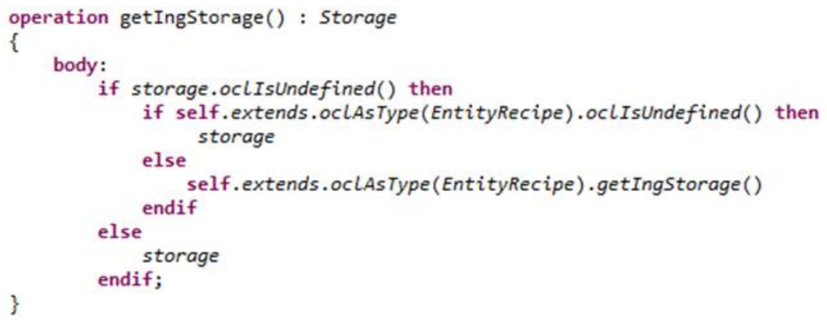
Figure 52: Validation during code generation

7.Implementation

The high level architecture and the detailed design of the repository generation tool are discussed in Chapters 5 and 6 consecutively. The purpose of this chapter is to describe the most important implementation aspects of the repository generation tool.

7.1 Implementation Model Language

The metamodel of the implementation model language is realized in Ecore. Ecore is the standard modeling language provided by the Eclipse Modeling Framework. The specialized getter APIs in the implementation model are implemented in the OCLinEcore Language [15]. The code snippet in Figure 53 shows how the getIngStorage() getter API is implemented in OCLinEcore. This getter API realizes the extension rules of EntityRecipes. If storage is not defined in the recipe itself, this getter API looks for storage in the recipes that this recipe extends. These rules are explained in the design of the implementation model language in Chapter 6. If storage is not defined and if the recipe does not extend another recipe, storage with default value of oclundefined is returned. This situation is checked in the generators and validators by using the oclIsUndefined () method of OCL.



```
operation getIngStorage() : Storage
{
    body:
        if storage.oclIsUndefined() then
            if self.extends.oclAsType(EntityRecipe).oclIsUndefined() then
                storage
            else
                self.extends.oclAsType(EntityRecipe).getIngStorage()
            endif
        else
            storage
        endif;
}
```

Figure 53: Code Snippet of the getIngStorage() getter API in OCLinEcore.

The rest of the specialized getter APIs are implemented in a similar fashion.

7.2 Implementation Model Editor

As discussed in Chapter 6, during the design of the implementation model language editor, EMFText is selected as the textual concrete syntax specification language. Therefore, the concrete syntax of the implementation model language is specified in EMFText, as shown in Appendix 1. Keywords are represented as strings. Pascal Casing (upper camel casing) is adopted as a standard for writing implementation models. Attributes are an exception to this standard. Attributes are written in lower camel casing. Pascal Casing is chosen because of the relative familiarity of stakeholders with this standard.

7.3 Repository Generator

As explained in the design of the repository generator component in Chapter 6, Acceleo is selected as the model-to-text (M2T) transformation language. Two elements of the Acceleo language are used. These elements are templates and queries. Templates are used to generate the code and queries are used to encapsulate complex expressions that manipulate model elements. Templates and queries are contained in a module. A module can import other modules. Acceleo allows calling plain Java services from templates and queries. For traceability reasons, Acceleo strongly recommends using Java services from queries instead of templates [16]. Let expression is the other Acceleo language construct that has been used extensively.

7.4 Implementation Model Wizard

This component is implemented in Acceleo and Java. The functionality of the implementation model wizard that generates the initial imp-models is implemented in Acceleo. The functionality of the implementation model wizard that prevents overwriting files containing implementation models unintentionally is implemented in Java. Plain Java services are used to find names and locations of models during implementation model generation. These java services are called from queries as recommended by Acceleo.

7.5 Implementation Model Validation

As discussed in Chapter 6, Acceleo templates and/or queries are selected as the implementation language for realizing the implementation model validation component. Since Acceleo queries are better than Acceleo templates in performance, they are used extensively. The results of a query evaluation are computed once and stored in a cache. This stored evaluation result is used when the query is called several times with the same set of parameters [16]. Java services are used to clean old validation results before new results can be displayed.

8.Testing

The purpose of this chapter is to discuss the testing techniques applied in this project.

8.1 Acceptance testing

Acceptance tests are provided at different levels to ensure that the repository generation tool is built according to the specifications. Each requirement is tested according to the test plans presented in Chapter 4. In addition to these tests, several fine-grained tests are identified during the realization phase of the requirements. These fine-grained tests resulted in various useful unit tests.

The test plans presented in Chapter 4 for each requirement are executed during five main test activities.

8.1.1. Review

The first line of defense to ensure the quality of the repository generation tool and the generated code was reviewing results. Results were reviewed by the supervisors and stakeholders at all phases of the project development process. During the planning session, several fine-grained acceptance tests are identified for each feature that must be verified through manual review and inspection. These acceptance tests are identified together with the stakeholders and supervisors. These tests must be satisfied to an acceptable level before a feature can be considered for further tests.

8.1.2. Implementation Model Validation

The goal of this test is to ensure that predefined data modeling rules are not violated. Implementation models must be validated without errors before code generation can be performed. The Implementation Model Validation rules are described in Section 6.8 in detail.

8.1.3. Build Tests

The goal of this test was ensuring that the generated code can be built successfully on the ASML build environment. This acceptance test must be satisfied before a feature can be considered for further tests.

8.1.4. Test Cases

Test cases were designed covering the major features of the repository generation tool that stakeholders use on their daily work. These test cases are identified together with supervisors and stakeholders. These test cases are summarized in Table 13.

Table 13: Test Cases for the generated repositories

Test Case	Motivation
SettingAndGettingPrimitiveTypeAttributes	This test case is intended to check the getters and setters of the primitive Type attributes, such as double and string, in the generated code.
SettingAndGettingValueObjectAttributes	This test case is intended to check the getters and setters of the ValueObject Type attributes in the generated code.
ValueObjectCopyConstructor	Since ValueObjects can be copied, this test case is intended to check the copy constructor of ValueObjects in the generated code.
ValueObjectAssignmentOperator	Since ValueObjects are assignable, this test case is intended to check the assignment operator of ValueObjects in the generated code.
AddingNewEntity	This test case is intended to ensure that Entity instances can be added successfully to a generated repository.
AddingExistingEntityThrows	This test case is intended to ensure that users are not trying to add existing Entity instances to generated repositories.
RemovingExistingEntity	This test case is intended to ensure that existing Entity instances can be removed successfully from a generated repository.

RemovingNonExistingEntityThrows	This test case is intended to ensure that users are not trying to remove non-existing Entity instances from generated repositories.
GettingExistingEntity	This test case is intended to ensure that existing Entity instances can be cloned successfully from a generated clone oriented repository.
GettingNonExistingEntityThrows	This test case is intended to ensure that users are not trying to clone non-existing Entity instances from generated clone oriented repositories.
UpdatingLocalEntityDoesNotAffectRepo	This test case is intended to ensure that updating a local clone of Entity instances does not affect the Entity instances in generated repositories.
UpdatingRepo	This test case is intended to ensure that updating Entity instances in the generated repositories is possible.

The test cases described in Table 13, are implemented in googletest [17]. Due to the familiarity of the stakeholders with this framework, googletest was selected as a testing framework in this project.

8.1.5. Release Tests

This test activity refers to uncovering bugs that are not detected or not covered by the previous test activities. These tests are performed by stakeholders after every new release of the repository generation tool during their daily work. Identified bugs are communicated to the author immediately. These bugs are considered in the planning session of the upcoming iteration and fixed according to their priority.

8.2 Regression Testing

The test cases identified for acceptance testing are finally added to regression tests which are provided to support evolution of the repository generation tool in the future. These regression tests ensure that previously provided features are not broken or new bugs are not discovered after introducing new changes to the repository generation tool.

8.3 Requirements Revisited

The requirements presented in Chapter 4 are revisited in order to determine whether they are satisfied or not. The acceptance tests specified by the test plan in each of the requirements are executed. If the test passes, the requirement is considered to be satisfied. The must have requirements for this project are all satisfied, as shown in Table 14.

Table 14: Must Have Requirements (MReq) – revisited

Requirement ID	Tests (pass/fail)	Satisfaction Level
MReq 1	**Pass**	**Satisfied**
MReq 2	**Pass**	**Satisfied**
MReq 3	**Pass**	**Satisfied**
MReq 4	**Pass**	**Satisfied**
MReq 5	**Pass**	**Satisfied**
MReq 6	**Pass**	**Satisfied**
MReq 7	**Pass**	**Satisfied**
MReq 8	**Pass**	**Satisfied**
MReq 9	**Pass**	**Satisfied**
MReq 10	**Pass**	**Satisfied**
MReq 11	**Pass**	**Satisfied**
MReq 12	**Pass**	**Satisfied**
MReq 13	**Pass**	**Satisfied**
MReq 14	**Pass**	**Satisfied**

The nice to have requirements for this project are shown in Table 15. NReq 1 is partially satisfied. NReq 3 is fully satisfied. NReq 3 is satisfied by deploying the repository generation tool in a Luna version of the Eclipse

Modeling Framework in a standalone ASML computer, as described by the test plan in Chapter 4. Because of time limitations, NReq 2 and NReq 4 are not satisfied.

Table 15: Nice to Have Requirements (NReq) – revisited

Requirement ID	Tests (pass/fail)	Satisfaction Level
NReq 1	**Fail**	**Partially Satisfied**
NReq 2	**Fail**	**Not Satisfied**
NReq 3	**Pass**	**Satisfied**
NReq 4	**Fail**	**Not Satisfied**

The identified won't have requirements are not satisfied within the scope of this project. However, to ensure that the architecture will not hamper realization of these requirements in the future, they are considered during the design of the repository generation tool. These considerations are presented in Table 16.

Table 16: Won't have Requirements (WReq) – revisited

Requirement ID	Remark
WReq 1	During the design of the Implementation Model Language, database repositories are represented as one of the storage kinds. In the ingredients metamodel shown in Figure 29, database is modeled as an abstract subtype of the storage ingredient. To allow the generation of database repositories in the future, the generation templates provide extension points that can be extended easily.
WReq 2	The remarks for **WReq 1** apply to this requirement as well.
WReq 3	The remarks for **WReq 1** apply to this requirement as well.

9.Conclusions

The purpose of this chapter is to conclude this report with a brief summary of the results obtained and the future work.

9.1 *Results*

The repository generation tool is built based on a flexible architecture in which domain models are decoupled from technology and implementation choices. The tool consists of an Implementation Model Language that helps users specify choices of implementation patterns without polluting their domain models with implementation details. To maximize flexibility, this language is based on the recipe-ingredient approach in a cookbook. To maximize productivity and facilitate learning the Implementation Model Language and its syntax, the tool contains an Implementation Model Wizard capable of creating initial implementation models from domain models. To early discover errors in the implementation model before code generation, the tool is equipped with an Implementation Model Validation. This protects the tool from producing a code that does not compile or a wrong code that compiles. The tool consists of a repository generator component to allow generation of repositories from domain models based on the recipes in implementation models. This is realized by providing several code generation modules.

The repository generation tool is equipped with models and unit tests for regression testing. These models and tests will be useful while adding new features to the tool in the future.

The results obtained in this project are directly being used by a metrology project within the ASML Metrology group. According to the feedback we have received from the metrology group, their productivity is significantly improved. They have already generated 600+ files of C++ code using the tool. Manipulation of domain models is very easy with the repository generation tool. The effort and time required to see changes in a domain model reflected in the code is reduced to a one button click.

In addition to the repository generation tool, an extensive requirements gathering and analysis was conducted for the Domain Model Language and its accompanying editor. The result of this analysis is shown in Appendix 5. These requirements were found to be useful while maturing the ASML data modeling environment which is used for defining domain models.

9.2 *Future Work*

The most obvious interesting additions to this tool are the requirements in the nice to have category which are not satisfied within the scope of this project. Due to time limitations, only a partial implementation of the boost intraprocess reference-oriented repositories is realized within the scope of this project. To provide a better performance alternative to the clone-oriented repositories, this partial implementation should be completed. In addition to this, no implementation is given for the boost interprocess reference-oriented repositories. This should also be completed. Furthermore, generating code comments would improve maintenance and diagnostics of the repository generation tool.

Some features of the domain models are not supported by the repository generation tool because of time limitations. Although these features are not critical to the stakeholders at the moment, we believe that these features should be implemented to provide a full-fledged code generation tool. The features that should be supported by the repository generation tool in the near future are summarized in Table 17.

Table 17: Summary of features supported by the code generator.

Category	**Features**	**Should be supported**
Implementation Model /Ingredients	Orientation	Reference
Domain Model	Entity	Non-Volatile
		Immutable
		Elements from other models, separate namespace.
	EntityMultiplicity	Constant values
	ValueObject	Elements from other models, separate namespace.
	Multiplicities	Different from [Zero or more]

		Ordering collections

Additionally, because of time limitations, regression tests are provided for only the heap based implementation of repositories. In order to provide a full-fledged regression testing framework, tests should also be provided for the boost interprocess repositories.

Furthermore, the requirements in the won’t have category shown in Table 16 should also be realized. These requirements are prioritized lower than the other requirements because of time limitations. However, these requirements are relevant and will be needed in the future.

10. Project Management

This chapter presents the project management strategies applied in this project.

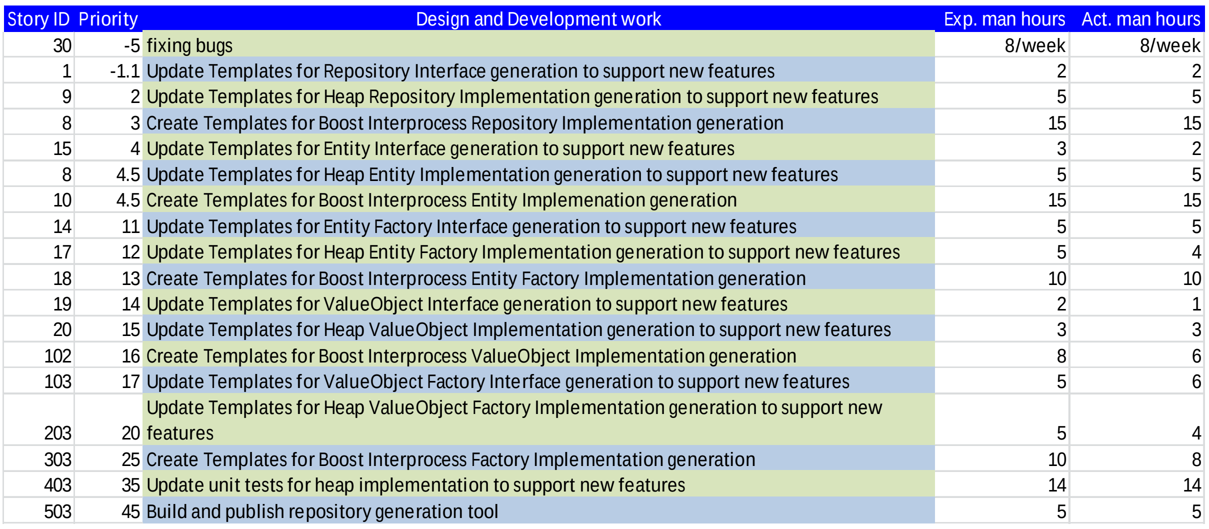
10.1 Approach

Early during the requirements identification and analysis process, it became apparent that requirements were changing over time. The priorities of requirements were also changing quite often. The conclusion was that sequential approaches, such as the waterfall, are not fitting for this project. Therefore, it was necessary to select an iterative development approach that can handle those changing requirements. The agile methodology, namely the SCRUM was found to be suitable in this project. However, since following the full SCRUM by the book was a heavyweight process for a one person team, a personalized lightweight version of the SCRUM was applied in this project. For example, the author was the project manager, the SCRUM master, and the SCRUM team all by himself. Daily standups were also not practical.

In our lightweight SCRUM, there were two sprints: a long sprint of three weeks and a short sprint of one week. The development was based on iterations. Iteration begins at the start of each long sprint and ends at the end of each long sprint. Intermediate results are demonstrated to and discussed with supervisors and stakeholders at the end of each short sprint. Project steering group (PSG) meetings were held at the end of each long sprint.

Iterations begin with a planning meeting together with the stakeholders and supervisors. In this meeting, features are identified based on the requirements and a backlog is filled out and prioritized, as shown in Figure 54. Results are discussed with stakeholders and supervisors at the end of every short sprint. At the end of the iteration, features are integrated, tested, and a new version of tool is released. Results are demonstrated to the PSG and the stakeholders. This way of working allowed stakeholders to use the tool on their daily work starting from the first iteration. To allow stakeholders request new features and report problems easily, an additional online backlog was also maintained. The feedback from stakeholders based on their daily work was found to be useful during this development process. Based on our experience during the first couple of iterations and the feedback from stakeholders, we concluded that this iterative pattern was suitable for this project. Therefore, this pattern was applied in all iterations during the entire development process.

Iterations focused on different activities based on a project plan. For example, the ninth iteration, tasks shown in Figure 54, focused on realizing the Repository Generator component in our reference architecture.

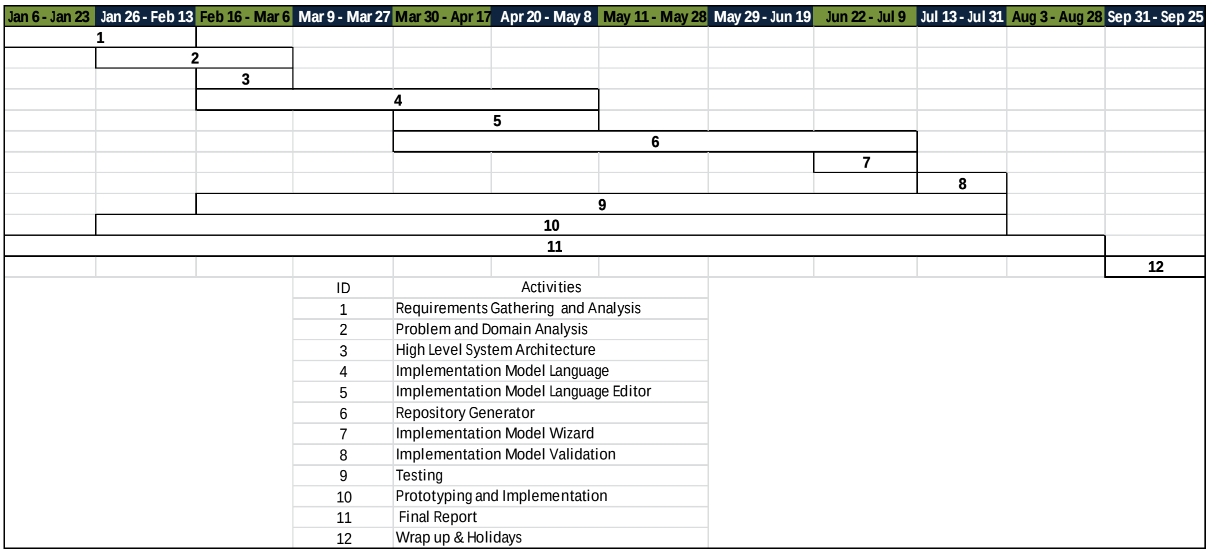


Story ID	Priority	Design and Development work	Exp. man hours	Act. man hours
30	-5	fixing bugs	8/week	8/week
1	-1.1	Update Templates for Repository Interface generation to support new features	2	2
9	2	Update Templates for Heap Repository Implementation generation to support new features	5	5
8	3	Create Templates for Boost Interprocess Repository Implementation generation	15	15
15	4	Update Templates for Entity Interface generation to support new features	3	2
8	4.5	Update Templates for Heap Entity Implementation generation to support new features	5	5
10	4.5	Create Templates for Boost Interprocess Entity Implemenation generation	15	15
14	11	Update Templates for Entity Factory Interface generation to support new features	5	5
17	12	Update Templates for Heap Entity Factory Implementation generation to support new features	5	4
18	13	Create Templates for Boost Interprocess Entity Factory Implementation generation	10	10
19	14	Update Templates for ValueObject Interface generation to support new features	2	1
20	15	Update Templates for Heap ValueObject Implementation generation to support new features	3	3
102	16	Create Templates for Boost Interprocess ValueObject Implementation generation	8	6
103	17	Update Templates for ValueObject Factory Interface generation to support new features	5	6
203	20	Update Templates for Heap ValueObject Factory Implementation generation to support new features	5	4
303	25	Create Templates for Boost Interprocess Factory Implementation generation	10	8
403	35	Update unit tests for heap implementation to support new features	14	14
503	45	Build and publish repository generation tool	5	5

Figure 54: Backlog snippet for repository generation, from the ninth iteration

10.2 *Project Planning*

To deliver the expected results, twelve main coarse-grained project activities were identified and executed according to the project plan shown in Figure 55. This project plan is the result of continuous evolution throughout the course of the project. The coarse-grained activities shown in the project plan are broken down into features during the planning session of the iterations in which they are executed. Features are identified and prioritized together with supervisors and stakeholders, as shown in Figure 54.



ID	Activities
1	Requirements Gathering and Analysis
2	Problem and Domain Analysis
3	High Level System Architecture
4	Implementation Model Language
5	Implementation Model Language Editor
6	Repository Generator
7	Implementation Model Wizard
8	Implementation Model Validation
9	Testing
10	Prototyping and Implementation
11	Final Report
12	Wrap up & Holidays

Figure 55: Project plan

To help project management decisions and to keep track of the amount of work that needs to be completed with respect to the remaining available time, the expected and actual efforts are recorded, as shown in Figure 54. In addition to this, the expected and actual development velocities are also tracked, as shown in Figure 56. It can be seen that during the first three sprints, the expected velocity exceeded the actual velocity. This was justified by the fact that the author had to study the problem domain extensively. From sprint 4 to sprint 6, it was possible to precisely predict the development velocity. This effort estimation and planning information is used as an input for project management decisions and for planning the upcoming iterations. Based on the input from this effort estimation, discussions with supervisors, and the feedback form stakeholders, an important project management decision was taken at the end of sprint seven. This decision was collocation of the project with the main stakeholders starting from the eighth sprint. This decision had a positive impact on the quality and speed of the project as also reflected in the chart shown in Figure 56. We can see that during sprints eight and nine, the actual development velocity exceeded the expected velocity. This was justified by the effective communication with the main users of the tool.

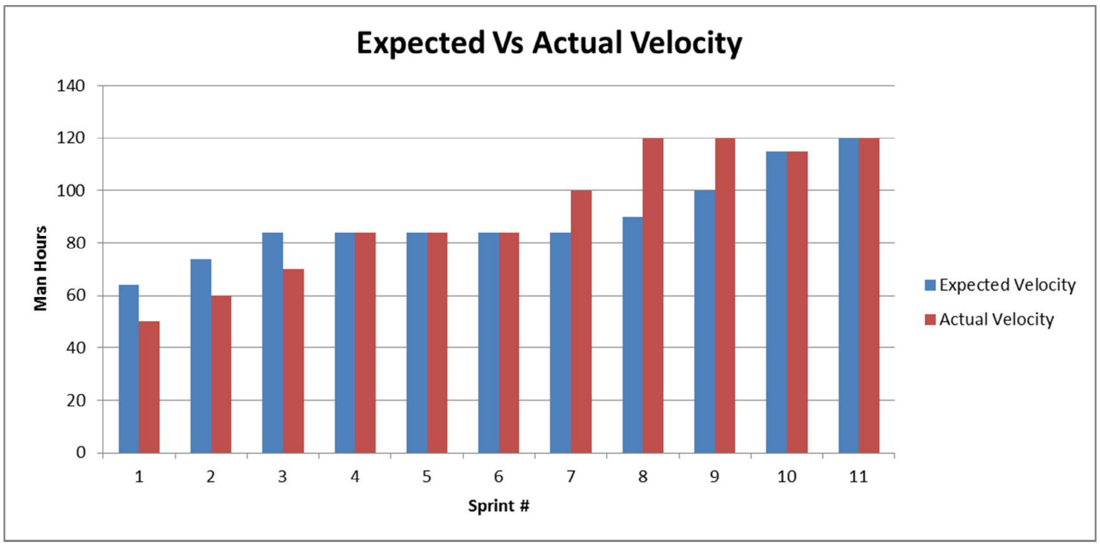


Figure 56: Expected Vs Actual Velocity in man hours, for the total of 11 long-sprints in this project

10.3 Risk Management

Over the course of this project, several potential risks that could have a negative impact on the project were identified and properly handled. These risks were identified through discussions with supervisors, brainstorming, and careful observation on daily bases. The most important ones are: Lack of resource, Lack of domain knowledge, priority of requirements, and communication with stakeholders.

i. **Lack of resources**
Time was the most expensive resource in this project. This project had a fixed time budget of nine months. Unless planned property, shortage of time could have a severe negative impact on the results obtained in this project. To help reduce the impact of this risk, proper effort estimation and planning was in place at different levels of granularity. Time was also considered when taking design decisions. For example, the time required to gain the necessary expertise was considered while selecting certain technologies.

ii. **Lack of domain knowledge**
Since there was no centralized domain knowledge base, the necessary domain knowledge is collected by talking to people, reading slides, and prototyping. Following introduction courses of one week was also found to be useful.

iii. **Priority of requirements**
The project started with a wider scope than what is delivered in the end. Considering the limited time budget, it was necessary to prioritize requirements according their importance. Therefore, together with the supervisors and stakeholders, the requirements that have the highest value to ASML were identified and prioritized higher than others. In spite of this extensive analysis and periodization, stakeholders continued prioritizing several requirements differently. Misunderstanding in the early communication with the stakeholders and the need to adjust the scope of the project were two main reasons for these changing requirements and their prioritization. To reduce the severity of this risk, an iterative development approach was selected. This approach allowed the priorities of requirements to change every sprint whenever necessary. To improve misunderstanding in the communication with stakeholders various project management decision were taken. These decisions are discussed in the next Section.

iv. **Communication with stakeholders**
Communication with stakeholders is another important risk that could have a negative impact on the results of this project. Since the project was originally located at the supervisors' workplace, there was a close communication and cooperation with the supervisors on daily bases. The communication with the stakeholders was mostly arranged on request and through emails. However, this communication was not enough. To minimize this communication problem, the main stakeholders were invited to participate in the weekly meetings. This arrangement improved the communication between the author, the supervisors, and the stakeholders. Unfortunately, even this communication was not enough. The ultimate solution to this communication problem was collocation of the project with the stakeholders. This decision had a positive impact on the results obtained and as a consequence on the satisfaction of the stakeholders.

11. Project Retrospective

This chapter reflects on the technical and organizational aspects of this project. The design criteria that guided the design process are also revisited.

11.1 Reflection

This project realizes a repository generation tool for generating repository implementation and access interfaces from domain models. In the course of this project, several technical and organizational lessons are learned.

In theory, software design should be implementation language independent. In practice, we learned that implementation technology influences greatly the design. Without considering the implementation technology, the Visitor/Decorator based design of the Repository Generator component suits better than the selected straightforward approach. However, the Visitor/Decorator based approach did not fit the selected template language, Acceleo. Therefore, the technical rule of thumb is that before creating your architecture and design, it is necessary to select the implementation languages that will be used to realize the system.

Communication with the stakeholders is one of the most important organizational lessons learned in this project. As explained as part of the project management, close communication with the stakeholders was determinant to the success of this project. Therefore, the organizational rule of thumb is that in any project it is necessary to maintain a close communication with the main stakeholders of the project. Emails and casual meetings are never enough.

11.2 Design opportunities revisited

In this project, several tradeoff architectural and design decisions are made to tackle the identified design opportunities and challenges. These design opportunities and challenges are: flexibility, reusability, and scalability.

- **Flexibility**

Flexibility is realized by providing the adapted MDA-based architecture in which domain models are decoupled from implementation and technology specific details. This architecture allows engineers develop domain models without polluting their domain models with repository implementation and technology choices for code generation. Implementation Modeling Language is provided to allow engineers select repository generation details and the domain model elements for which those choices are applicable. To maximize flexibility, the Implementation Model Language is designed based on the recipe-ingredient approach. Repository generators are provided for the C++ language.

- **Reusability**

Realizing code generation with a minimum effort is achieved by providing mechanisms to reuse implementation model artifacts. We realize this by providing language features for importing external models, extending existing recipes, and reusing ingredients. Section 6.1 explains the feature for importing other implementation models. The feature for extending an existing recipe is discussed in Section 5.7.3. Ingredients are explained in Section 5.7.2. These ingredients can be reused while defining multiple implementation models.

- **Scalability**

At the modeling level, scalability is realized by mapping one domain model to one or more implementation models. In this way, the architecture can handle arbitrarily large domain models. This is discussed in Chapter 5, Section 5.8 in detail. At the generation level, the templates and queries are implemented in such a way that execution time increases linearly with the number of elements in a model.

Glossary

PDEng stands for the Professional Doctorate in Engineering degree at the Eindhoven University of Technology (TU/e).
SW is an abbreviation for Software.
HW is an abbreviation for Hardware.
MReq represents the Must Have Requirements of the MoSCoW requirements specification technique.
NReq represents the Nice to Have Requirements of the MosCow requirements prioritization technique.
WReq represents the Won't Have Requirements of the MosCow requirements prioritization technique.
MReq represents the Must Have Requirements of the MoSCoW requirements specification technique.
MDA is an abbreviation for Model Driven Architecture.
OMG stands for Object Management Group.
PSG stands for project steering group
EPS stands for Element Performance Specification (EPS), which is a document used within ASML to describe the functional, performance and non-functional (e.g. reliability) requirements of a (sub) system.
OCL stands for Object Constraint Language.
Implementation Model an Implementation Model or imp-model for short, or a Generator Model is a model containing recipes and ingredients.
Domain Model represents the domain model of the TWINSCAN with respect to data.
Intraprocess is the local memory or Heap based implementation of repositories.
Interprocess is the shared memory implementation of repositories.

Bibliography

[1] “About ASML.” .

[2] K. Van Hee and K. Van Overveld, “New criteria for assessing a technological design,” no. April, 2012.

[3] E. Evans, “Domain-Driven Design,” vol. 7873, no. 415, 2003.

[4] DSDM, “MoSCoW Prioritisation.”

[5] A. Cockburn, “Writing effective use cases,” *Work*, 2001.

[6] P. Kruntchen, “Architectural blueprints–the” 4+ 1” view model of software architecture,” *IEEE Softw.*, vol. 12, no. November, pp. 42–50, 1995.

[7] A. B. Ormsc, C. Burt, D. Dsouza, K. Duddy, W. El Kaim, W. Frank, S. Iyengar, J. Miller, J. Mischkinsky, J. Mukerji, J. Siegel, R. Soley, S. Tyndal-, A. Uhl, A. Watson, and B. Wood, “Model Driven Architecture (MDA) Document number ormsc / 2001-07-01,” *Architecture*, pp. 1–31, 2001.

[8] “Eclipse Modeling Framework, documents.”

[9] “Boost.”

[10] E. Gamma, R.Helm, R. Johnson, and J. Vlissides, “Design Patterns,” 2007.

[11] “EMFText User Guide,” 2012.

[12] “Xtext.”

[13] Obeo, “Acceleo documentation.”

[14] “Xtend documentation.”

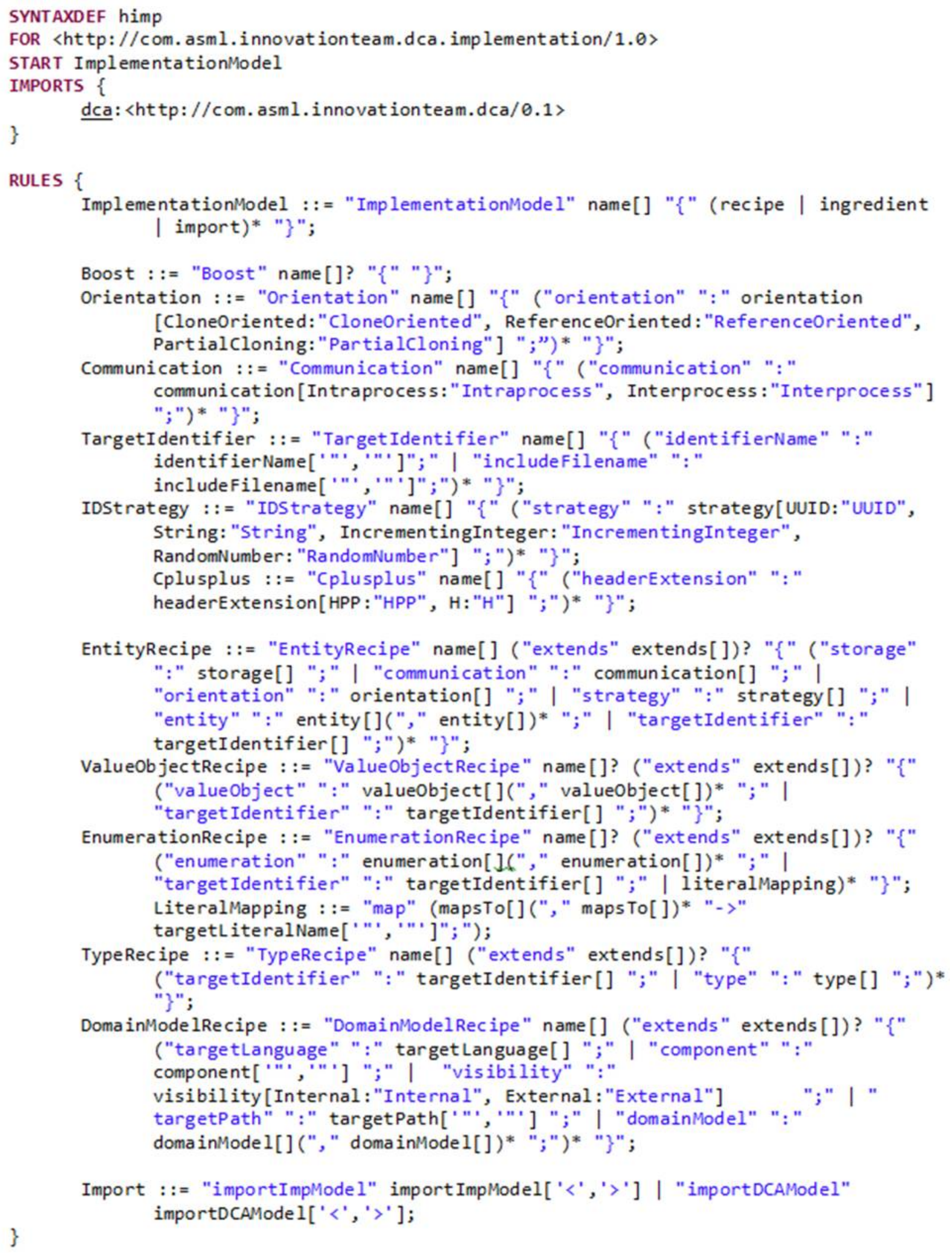
[15] “OCLinEcore.”

[16] “Acceleo Documentation.”

[17] " googletest"

Appendix 1 – Implementation Model Concrete Syntax

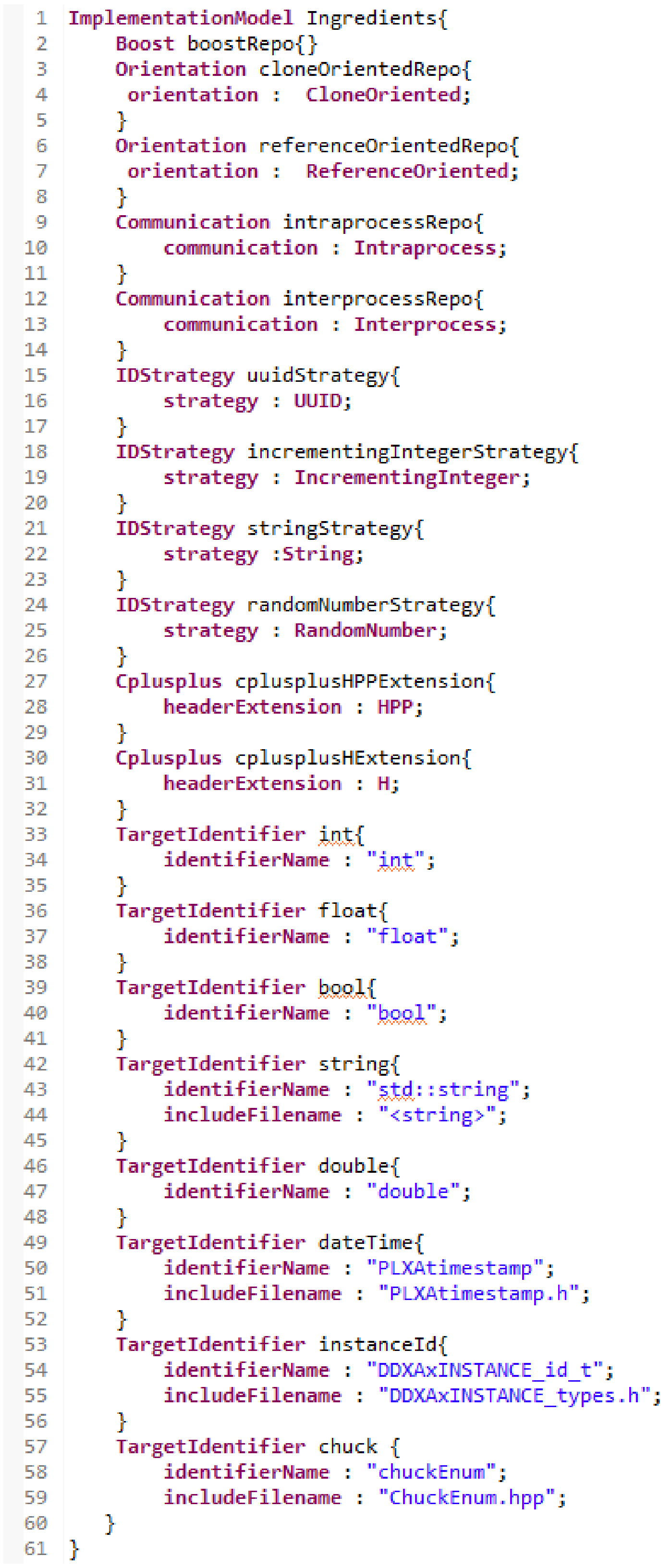
The concrete syntax of the Implementation Model Language is specified in EMFText. Keywords are represented as quoted strings. The rest of the elements are metaclasses in the abstract syntax of the language. This is shown in Figure 57.



```
SYNTAXDEF himp
FOR <http://com.asml.innovationteam.dca.implementation/1.0>
START ImplementationModel
IMPORTS {
        dca:<http://com.asml.innovationteam.dca/0.1>
}

RULES {
        ImplementationModel ::= "ImplementationModel" name[] "{" (recipe | ingredient
                | import)* "}";

        Boost ::= "Boost" name[]? "{" "}";
        Orientation ::= "Orientation" name[] "{" ("orientation" ":" orientation
                [CloneOriented:"CloneOriented", ReferenceOriented:"ReferenceOriented",
                PartialCloning:"PartialCloning"] ";”)* "}";
        Communication ::= "Communication" name[] "{" ("communication" ":"
                communication[Intraprocess:"Intraprocess", Interprocess:"Interprocess"]
                ";")* "}";
        TargetIdentifier ::= "TargetIdentifier" name[] "{" ("identifierName" ":"
                identifierName['"','"']";" | "includeFilename" ":"
                includeFilename['"','"']";")* "}";
        IDStrategy ::= "IDStrategy" name[] "{" ("strategy" ":" strategy[UUID:"UUID",
                String:"String", IncrementingInteger:"IncrementingInteger",
                RandomNumber:"RandomNumber"] ";")* "}";
                Cplusplus ::= "Cplusplus" name[] "{" ("headerExtension" ":"
                headerExtension[HPP:"HPP", H:"H"] ";")* "}";

        EntityRecipe ::= "EntityRecipe" name[] ("extends" extends[])? "{" ("storage"
                ":" storage[] ";" | "communication" ":" communication[] ";" |
                "orientation" ":" orientation[] ";" | "strategy" ":" strategy[] ";" |
                "entity" ":" entity[]("," entity[])* ";" | "targetIdentifier" ":"
                targetIdentifier[] ";")* "}";
        ValueObjectRecipe ::= "ValueObjectRecipe" name[]? ("extends" extends[])? "{"
                ("valueObject" ":" valueObject[]("," valueObject[])* ";" |
                "targetIdentifier" ":" targetIdentifier[] ";")* "}";
        EnumerationRecipe ::= "EnumerationRecipe" name[]? ("extends" extends[])? "{"
                ("enumeration" ":" enumeration[]("," enumeration[])* ";" |
                "targetIdentifier" ":" targetIdentifier[] ";" | literalMapping)* "}";
                LiteralMapping ::= "map" (mapsTo[]("," mapsTo[])* "->"
                targetLiteralName['"','"']";");
        TypeRecipe ::= "TypeRecipe" name[] ("extends" extends[])? "{"
                ("targetIdentifier" ":" targetIdentifier[] ";" | "type" ":" type[] ";")*
                "}";
        DomainModelRecipe ::= "DomainModelRecipe" name[] ("extends" extends[])? "{"
                ("targetLanguage" ":" targetLanguage[] ";" | "component" ":"
                component['"','"'] ";" |  "visibility" ":"
                visibility[Internal:"Internal", External:"External"]       ";" | "
                targetPath" ":" targetPath['"','"'] ";" | "domainModel" ":"
                domainModel[]("," domainModel[])* ";")* "}";

        Import ::= "importImpModel" importImpModel['<','>'] | "importDCAModel"
                importDCAModel['<','>'];
}
```

Figure 57: Implementation Model Concrete Syntax

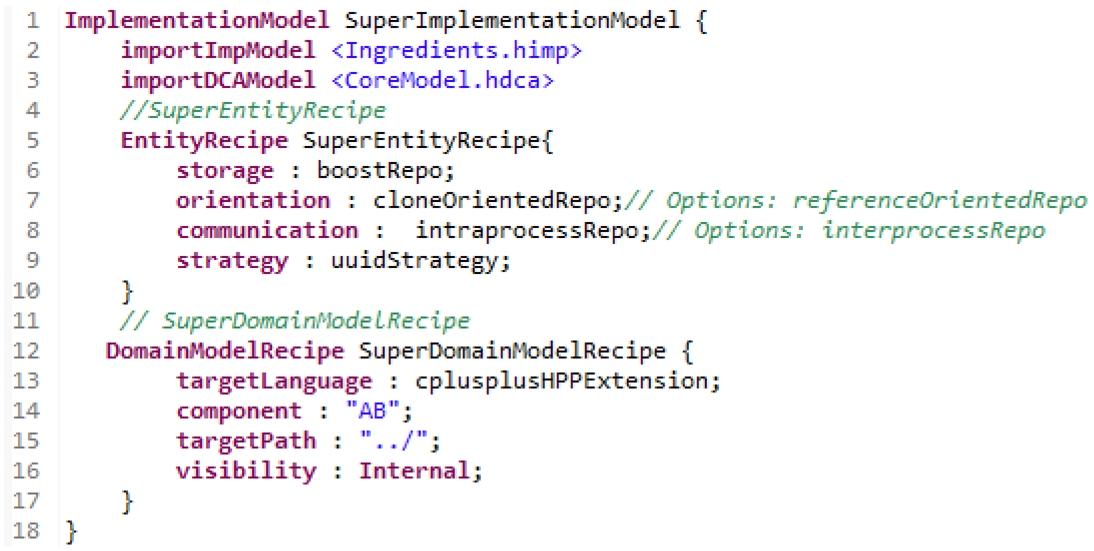
Appendix 2 – Implementation Model defining only ingredients



```
ImplementationModel Ingredients{
    Boost boostRepo{}
    Orientation cloneOrientedRepo{
     orientation :  CloneOriented;
    }
    Orientation referenceOrientedRepo{
     orientation :  ReferenceOriented;
    }
    Communication intraprocessRepo{
        communication : Intraprocess;
    }
    Communication interprocessRepo{
        communication : Interprocess;
    }
    IDStrategy uuidStrategy{
        strategy : UUID;
    }
    IDStrategy incrementingIntegerStrategy{
        strategy : IncrementingInteger;
    }
    IDStrategy stringStrategy{
        strategy :String;
    }
    IDStrategy randomNumberStrategy{
        strategy : RandomNumber;
    }
    Cplusplus cplusplusHPPExtension{
        headerExtension : HPP;
    }
    Cplusplus cplusplusHExtension{
        headerExtension : H;
    }
    TargetIdentifier int{
        identifierName : "int";
    }
    TargetIdentifier float{
        identifierName : "float";
    }
    TargetIdentifier bool{
        identifierName : "bool";
    }
    TargetIdentifier string{
        identifierName : "std::string";
        includeFilename : "<string>";
    }
    TargetIdentifier double{
        identifierName : "double";
    }
    TargetIdentifier dateTime{
        identifierName : "PLXAtimestamp";
        includeFilename : "PLXAtimestamp.h";
    }
    TargetIdentifier instanceId{
        identifierName : "DDXAxINSTANCE_id_t";
        includeFilename : "DDXAxINSTANCE_types.h";
    }
    TargetIdentifier chuck {
        identifierName : "chuckEnum";
        includeFilename : "ChuckEnum.hpp";
   }
}
```

Figure 58: Ingredients

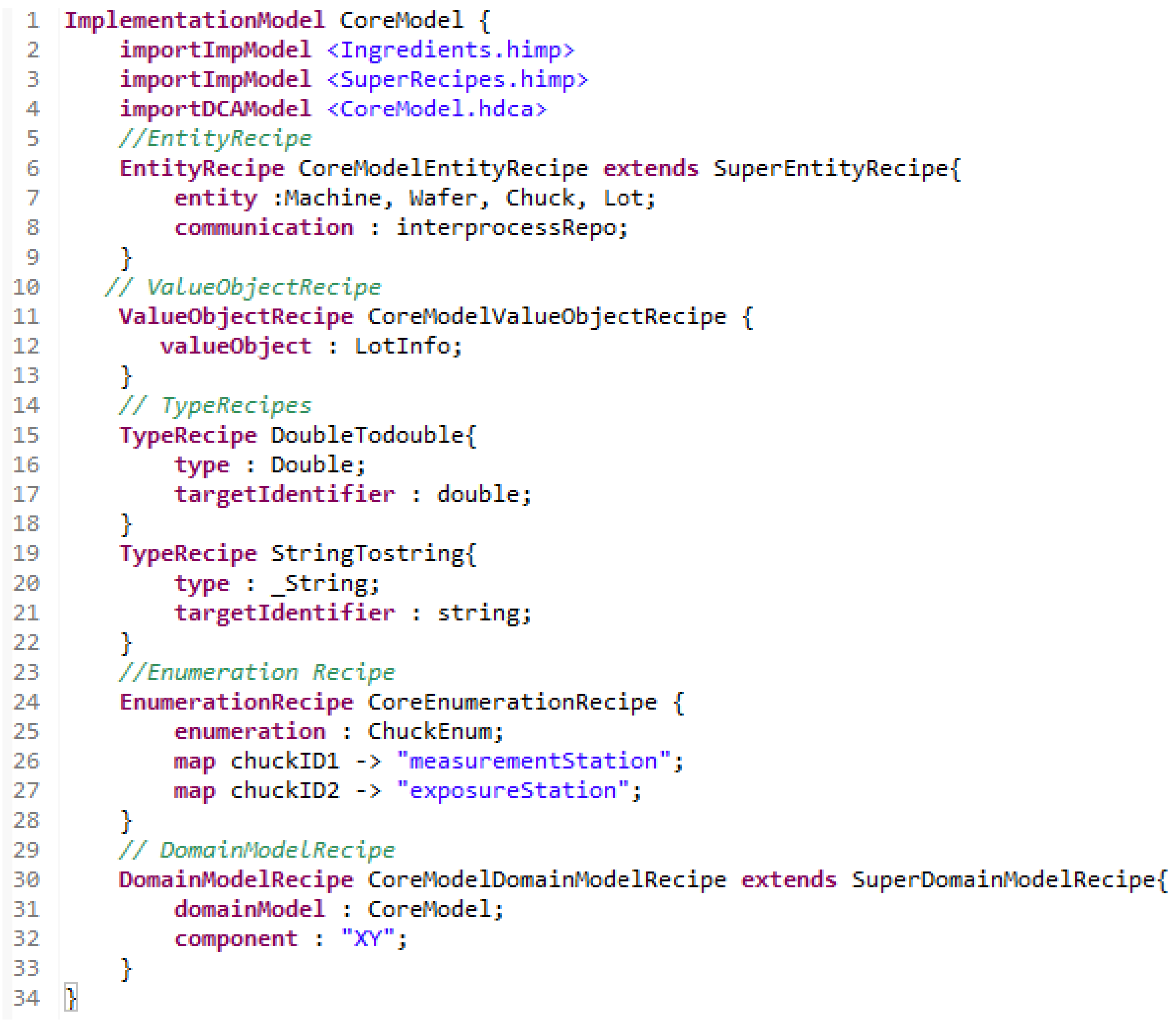
Appendix 3 – Implementation Model defining only super recipes



```
ImplementationModel SuperImplementationModel {
    importImpModel <Ingredients.himp>
    importDCAModel <CoreModel.hdca>
    //SuperEntityRecipe
    EntityRecipe SuperEntityRecipe{
        storage : boostRepo;
        orientation : cloneOrientedRepo;// Options: referenceOrientedRepo
        communication :  intraprocessRepo;// Options: interprocessRepo
        strategy : uuidStrategy;
    }
    // SuperDomainModelRecipe
   DomainModelRecipe SuperDomainModelRecipe {
        targetLanguage : cplusplusHPPExtension;
        component : "AB";
        targetPath : "../";
        visibility : Internal;
    }
}
```

Figure 59: super recipes that can be reused in other recipes

Appendix 4 – Implementation Model defining recipes for code generation



```
ImplementationModel CoreModel {
    importImpModel <Ingredients.himp>
    importImpModel <SuperRecipes.himp>
    importDCAModel <CoreModel.hdca>
    //EntityRecipe
    EntityRecipe CoreModelEntityRecipe extends SuperEntityRecipe{
        entity :Machine, Wafer, Chuck, Lot;
        communication : interprocessRepo;
    }
   // ValueObjectRecipe
    ValueObjectRecipe CoreModelValueObjectRecipe {
       valueObject : LotInfo;
    }
    // TypeRecipes
    TypeRecipe DoubleTodouble{
        type : Double;
        targetIdentifier : double;
    }
    TypeRecipe StringTostring{
        type : _String;
        targetIdentifier : string;
    }
    //Enumeration Recipe
    EnumerationRecipe CoreEnumerationRecipe {
        enumeration : ChuckEnum;
        map chuckID1 -> "measurementStation";
        map chuckID2 -> "exposureStation";
    }
    // DomainModelRecipe
    DomainModelRecipe CoreModelDomainModelRecipe extends SuperDomainModelRecipe{
        domainModel : CoreModel;
        component : "XY";
    }
}
```

Figure 60: An implementation model that can be used as a direct input for code generation

Appendix 5 – The Identified Requirements for the domain model language

Table 18: Identified Requirements for the domain model language

ID	Description, Rationale, and Testing	Ref.
MReq 1	*Description:* The domain model language must support the concepts: **MReq 1.1** Entities. **MReq 1.2** Value objects. **MReq 1.3** Inheritance. **MReq 1.4** Mutability. **MReq 1.5** Volatility. **MReq 1.6** Compositions and attributes. **MReq 1.7** Enumeration types. **MReq 1.8** Primitive types. **MReq 1.9** Constants. **MReq 1.10** Associations. **MReq 1.11** Multiplicities. These concepts are described in detail in Section Chapter 3. *Rationale*: Without the ability to create instances of these concepts, it will not be possible to create domain models. *Testing*: The ability to create instances of each of these concepts is tested by using a suitable testing framework. These concepts are tested manually. This is also used to test the editor in requirement **MReq2**.	ASML Architecture Group & ASML Metrology Group
MReq 2	*Description:* A graphical and textual editors must be provided for defining domain models by using the concepts presented in **MReq 1**. The graphical editor must also provide a proper distinction mechanism between the different concepts when they appear in a model as well as when they are printed in black and white. *Rationale*: Without an editor, the defining domain models will not be possible. *Testing*: The test plan in **MReq1** is used to also test the editor.	ASML Architecture Group & ASML Metrology Group
MReq 3	*Description:* The tool must support dependencies between multi-owner domain models. It should be possible to refer one model from another. In this situation the referenced model must stay unchanged. Proper interfaces need to be defined for granting access to these models based on the ASML's scopefiles. *Rationale*: without this feature, it would not be possible to compose models from different parts located at different functional clusters, building blocks, and components. *Testing*: This is tested by deploying models in two components from two different functional clusters in ASML's dumbo view. It should be possible to reference from one of the models to the other. The referenced model must stay unchanged.	ASML Architecture Group & ASML Metrology Group

NReq 4	*Description:* The domain model editor must support dragging and dropping domain model elements from other models into the editor canvas. This allows graphical visualization of dependencies between models. Any relationships with the model already in the editor should be displayed. *Rationale*: Without this feature, it would not be possible to easily visualize dependencies between models in the same or different functional clusters, building blocks, and software components. *Testing*: This is tested by creating models in two functional clusters, building blocks, and software components in the ASML's dumbo view. It should be possible to visualize interdependencies between models by dragging and dropping elements from one model to the other. The relevant associations must also be displayed in the editor.	ASML Architecture Group
MReq 5	*Description:* the tool must support diff and merge on the textual representation of the domain models. *Rationale*: Without this feature, it would not be easy to compare different versions of a model, view the differences, and merge models. *Testing*: This is tested by deploying a model in two release parts in the ASML's dumbo view. One or the two of these models is changed, and it should be possible to view the differences and merge the two models into a latest version.	ASML Architecture Group
MReq 6	*Description:* The tool must be deployable in the ASML's Eclipse-based WindRiver Workbench for Linux environment. Since ASML is moving towards the Eclipse Luna, the tool must be based on the Luna version of Eclipse. It should also be possible to use the tool outside the ASML WindRiver Workbench, possibly standalone on an ASML computer. *Rationale*: On one hand, the ASML software architects and software engineers use the Eclipse-based WindRiver Workbench as a development environment. If the tool cannot be deployed in the WindRiver Workbench, it will not be handy to be used by these architects and software engineers. On the other hand, there are software architects and software engineers who may use the tool for experimental purposes. If the tool cannot be used standalone on an ASML computer, these people will not be able to use it. *Testing*: The deployability of the tool in the ASML's Eclipse-based WindRiver workbench is tested by installing the tool in the workbench. This will also be sent to the ASML Software Development Environment Group for testing. The ASML Software Development Environment Group will send the set of plugins to WindRiver for a final check. The ability of the tool to be used standalone is tested by installing the tool in a separate eclipse on an ASML computer.	ASML Architecture Group & ASML Software Development Environment Group
MReq 7	*Description:* The solution must be scalable against the number of elements in a model. The tool must support at least 50 models each with 100 elements and 10 model imports. *Rationale*: Without this feature, it would not be possible to apply the solution to bigger models. *Testing*: This is tested by creating one domain model with 100 elements and 10 model imports and duplicate this model 50 times.	ASML Architecture Group
NReq 8	*Description:* the tool must be integrated with the ASML build system.	ASML Architecture

	Rationale: without the ability to integrate with the ASML build system, it would not be possible for the tool to work and be released seamlessly with the rest of the DCA tools. *Testing*: This feature is tested manually.	Group
NReq 9	*Description:* The domain model languages and editors must support the language concept comment for documentation purposes. *Rationale*: Without this comment, it may not be easy to add comments to models that are used during code generation. *Testing*: This requirement is tested by instantiating the comment language concept by using its editor.	ASML Metrology Group
NReq 10	*Description:* the domain model language may support specification of attributes in terms of their units, measurement types, and coordinate system. *Rationale*: Without this feature, it would not be possible to specify the attributes in terms of their units, types of measurements, and coordinate systems. *Testing*: This is tested manually by creating a model containing elements that have attributes that require specification of units, measurements, and coordinate systems. It must be possible to specify these attributes in terms of the units, type of measurement, and their coordinate systems.	ASML Metrology Group
NReq 11	*Description:* The domain and implementation model language may support the concept package. *Rationale*: Without this feature, it would not be possible to group model elements in one or more smaller manageable packages. *Testing*: This is tested manually by creating models containing multiple model elements in multiple packages.	ASML Metrology Group
NReq 12	*Description:* The domain model language and its editor may support the concept aggregate root. *Rationale*: Without the concept aggregate root, it would not be easy to define and maintain boundaries between related and unrelated domain model elements. *Testing*: This requirement is tested by instantiating the aggregate root language concept by using the provided editor.	ASML Metrology Group

About the Author

Tesfahun Tesfay received his BSc. degree in Information Technology from Mekelle Institute of Technology, Ethiopia in 2010. After graduation he worked for Defence Engineering College, Ethiopia, as a graduate assistant for one year. In September 2011 he came to the Netherlands and started his Master degree in Computer Science, Software Engineering track at the faculty of EEMCS-Electrical Engineering, Mathematics and Computer Science of the University of Twente, Enschede, from where he obtained his master degree in August 2013. During his internship at Sytematic Software, The Hague, The Netherlands, he developed a language for modeling REST APIs. During his master thesis at BiZZdesign, Enschede, The Netherlands, he developed an automated system, annotation language and algorithms for identifying business process distribution options and their consequences for cloud-based BPM. In 2013, Tesfahun joined the PDEng program in Software Technology at the Eindhoven University of Technology. He worked on several in-house projects from various companies. During his PDEng final project at ASML, Veldhoven, The Netherlands, he developed a repository generation tool that is used for generating repositories from domain models. His main interests include language engineering, model-driven engineering, model-driven architecture, software architectures, design patterns, and basketball and soccer.

3TU.School for Technological Design, Stan Ackermans Institute offers two-year postgraduate technological designer programmes. This institute is a joint initiative of the three technological universities of the Netherlands: Delft University of Technology, Eindhoven University of Technology and University of Twente. For more information please visit: www.3tu.nl/sai.