ISSUES OF ITERATIVE MDA-BASED SOFTWARE DEVELOPMENT PROCESSES

MASTER THESIS

Author: Geert Vos

Thesis for the masters degree Computer Science.
Department of Computer Science, University of Twente, the Netherlands.

Getronics PinkRoccade
Apeldoorn, 5-4-2008

Supervising committee:
dr. I. Kurtev (first supervisor)
prof. dr. ir. M. Akşit
dr. ir. K. G. van den Berg
A. Goknil, MSc.
ir. J.W. van Veen
drs. H. Nieboer

I TITLE PAGE

Title: Issues of iterative MDA-based software development processes

Author: Geert Vos BSc.
Student number: S0107670

MSc program: Computer Science
Track: Software Engineering
Institute: University of Twente, the Netherlands
Faculty: Electrical Engineering, Mathematics and Computer Science

Company: Getronics PinkRoccade
Fauststraat 3
7323 BA Apeldoorn
The Netherlands
http://www.getronicspinkroccade.nl

Date: 5-4-2008

Supervising committee:

dr. I. Kurtev (first supervisor)
University of Twente

prof. dr. ir. M. Akşit
University of Twente

dr. ir. K. G. van den Berg
University of Twente

A. Goknil, MSc.
University of Twente

ir. J.W. van Veen
Getronics PinkRoccade

drs. H. Nieboer
Getronics PinkRoccade

II ABSTRACT

Since the beginning of computer programming people have been raising the level of abstraction. The latest step in raising the level is probably Model Driven Engineering (MDE). In MDE, models are not only used to assist in the development process, instead they are the primary artifacts. Model Driven Architecture(MDA) is a specific form of MDE in which industry standards are adopted. MDA is based on the following standards: Unified Modeling Language (UML), XML Meta Interchange (XMI), Query/View/Transformation (QVT), Object Constraint Language (OCL) and Meta Object Facility (MOF).

Currently, there is a lack of knowledge and experience in applying MDA in an iterative development process. With this project we try to obtain that knowledge and experience by conducting a case study. In this case study an existing software system is rebuilt using the MDA approach and the Rational Unified Process (RUP). By rebuilding this system we tried to answer two questions, what are the critical issues of an iterative MDA-based development process and what are the critical issues with respect to maintenance of an MDA-based product.

The major parts of the case study are the rebuilding of a small-sized software system using RUP and MDA, and applying maintenance to that system. The first part of the case study consists of four phases: the inception phase, elaboration phase, construction phase and the transition phase. In the inception phase we developed an architecture for the system and an architecture for the MDA approach. Both architectures were tested in the elaboration phase. In this phase the first use-case was implemented. Two meta-models were created to implement this use-case, an ASP and a C# meta-model. We also created two UML profiles. A profile for the domain models and a profile for user-experience models. The meta-models and profiles were developed in the elaboration phase. Based on the UML profiles, two models were made. A domain model, modeling the business objects of our system and an user-experience model which models the user interface and navigation of our system. The UML models were transformed into C# and ASP models using model transformations written in QVT. In the construction phase the second use-case was implemented. The implementation required adaption of the meta-models and transformations. Also a new meta-model was introduced: Sitemap. This meta-model is used to model the main menu of the system at platform specific level. The last phase of this project, the transition phase, consisted of two iterations. In the first iteration the rebuilt system was tested and deployed. In the second iteration we applied maintenance to the system. We implemented four change requests to observe the impact of the MDA approach with respect to maintenance.

During the evaluation of this case study a number of issues were observed. The following issues were identified: “Model and meta-model co-evolution is difficult”, “Structural incongruence increases transformation complexity”, “Lack of object orientation in QVT”, “Transformation composition”, and “Model composition is difficult”. A description and a solution (if known) is provided for each issue.

We concluded that the current state of the practice is that MDA is certainly possible and can be applied in combination with an iterative development process. We concluded that current tools and languages support the basic features needed for MDA, but that problems can be expected when complexity of the models and transformations increases.

III ACKNOWLEDGEMENTS

During this project a number of people have been very helpful to me. One of them is Ivan Kurtev. He introduced me in the field of Model Driven Engineering, which lead to this assignment. As my first supervisor he also helped me during my project, during the process of writing and especially finding the structure for my thesis. I would also like to thank Harry Nieboer for the daily supervision at Getronics PinkRoccade. Thanks to Harry the scope of the project was kept narrow. He also helped me with details of the Rational Unified Process, one if his key expertise areas. I would also like to thank Jan Willem van Veen from Getronics PinkRoccade. Jan Willem was the person that arranged my internship at Getronics PinkRoccade and he helped me setting up my project. Of course I would also like to thank the other people from Getronics PinkRoccade who discussed the topic of MDA with me or who helped me with my project: Ad van Klaveren, Tim Abeln, Arash Nezami, Gert Veldhuizen van Zanten en Kees van Sighem. I would like to thank Klaas van den Berg and Arda Goknil from the University of Twente for their work on my thesis and for the support during my project. The last person I would like to thank is my girlfriend Suzanne Snellenberg for the patience, the understanding and for reading my work.

IV LIST OF ABBREVIATIONS

Abbreviation	Meaning
ASP	Active Server Pages
CIM	Computational Independent Model
CMOF	Complete MOF
CR	Change Request
DSL	Domain Specific Language
EBNF	Extended Backus Naur Form
EMF	Eclipse Modeling Framework
EMOF	Essential MOF
EPF	Eclipse Process Framework
GPL	General Purpose Language
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
OPENUP	Open Unified Process
OpenUP	Open Unified Process
ORM	Object Relational Mapper
PIM	Platform Independent Model
POC	Proof of Concept
PSM	Platform Specific Model
QVT	Query/View/Transformation
RUP	Rational Unified Process
SQL	Structured Query Language
UC	Use Case
UML	Unified Modeling Language
UX	User Experience
XMI	XML Metadata Interchange

V LIST OF FIGURES

VI LIST OF CODE FRAGMENTS

TABLE OF CONTENTS

Chapter 1

1 INTRODUCTION

This chapter provides an introduction to the thesis and the backgrounds of the project. This chapter creates a context for the project and states the problem statement and research questions.

1.1 BACKGROUND

Since the beginning of computer programming people have been raising the level of abstraction. The first assembler languages we used to simplify computer programming. Instead of directly programming in machine language, assembler raised the level of abstraction to instructions and mnemonics. The next big step in raising the level of abstraction was taken by John Backus in 1954. He and his team invented Fortran, which was the first high level programming language. In the 1960's the first Object Oriented programming languages appeared. Object orientation was again a new level of abstraction to simplify computer programming.

The latest step to raise the level of abstraction is probably Model Driven Engineering (MDE) (Kent, 2002). This is currently a topic that has the interest of researchers and companies. The promise of MDE to raise the level of abstraction in software development makes it an interesting topic. It is a common practice to create models to assist developers with the implementation of an application. These models are often used as documentation and as guidelines for development but are not directly involved in the production of an application. In MDE the models are not only used to assist development, but the models form the backbone of the development process. In the view of MDE everything is a model including the source code.

Many companies are currently interested in Model Driven Architecture (MDA) (Object Management Group, 2003) as an approach for applying MDE principles. MDA is proposed by the Object Management Group (OMG) and is supported by industry standards like UML (Object Management Group, 2007), MOF (Object Management Group, 2006), OCL (Object Management Group, 2003) and XMI (Object Management Group, 2007). MDA, as described by the OMG, is one of the methods to apply MDE.

This study is carried out in the context of the QuadRead project (Project Quadread, 2008). This project is a joint research project from the University of Twente and industrial partners. The aim of the project is a better alignment of analysts and architects. The academic partners provide the project with research questions and the industrial partners provide case studies to strengthen practical applicability. Getronics PinkRoccade is one of the industrial partners of this project.

Getronics PinkRoccade is one of the larger players on the software market and provides services ranging from workplace management to the development, installation and maintenance of software systems. This project is carried out within the business unit Microsoft Application Services. This business unit is specialized in the development and maintenance of software systems based on Microsoft products and technology.

1.2 PROBLEM STATEMENT

Getronics PinkRoccade is interested in applying MDA in practice, but there are many problems that needs to be solved before MDA can be put to practice. Getronics PinkRoccade has done research on MDA in the past and also has a department working on model driven engineering. Companies like Getronics PinkRoccade often use iterative software development processes to minimize the risk in software development. At the start of this project we held a problem analysis session for the following problem: "**MDA is not applied on a large scale in the software industry**". Starting from this general problem the stakeholders of this project identified a number of causes for this problem. The results from the session were send to all stakeholders. Each of the stakeholders was asked to divide 10 points over the different causes. The highest ranked causes were selected as input for this project:

- MDA is not iterative
- Lack of experience with MDA and maintenance
- It is unclear whether models are easier to maintain than source code

These problems are the motivation for this master project. The results of the problem analysis session are available in Appendix A.

1.3 RESEARCH QUESTIONS

There is currently no commonly accepted software development process to apply model driven development. Some effort has been made to create a development process (Eclipse Process Framework OpenUP/MDD, 2006). There is also some work done on combining MDA with the Rational Unified Process (Brown & Conallen, 2005). Still there is no common agreement on what development process to use. A development process like RUP is an iterative development process and nowadays considered a best practice by many companies. Combining MDA with RUP seems a logical step, but it is unclear if this works in practice. A closer look on MDA reveals a more classic waterfall software development process (Wegener, 2002). This contradicts with the iterative approaches we use today.

Another interesting question is related to maintenance. The promise of MDA is to raise the level of abstraction. This would both lead to faster software development as an increase in maintainability. The general idea is that models are easier to maintain then code. Instead of changing 10 scattered lines of code, changing a single property in a model seems easier. But what are the issues regarding to MDA and maintenance?

The following two research questions forms the basis for this study:

"**What are the critical issues in an iterative MDA-based software development process?**"

"**What are the critical issues with maintenance activities in MDA-based software development?**"

With critical issues we mean issues that will lead to the failure of this project. In this sense, issues can be considered non-critical for this project but they may pose a threat to other projects.

1.4 APPROACH

There is currently little knowledge about how MDA can be applied in practice and what issues will be encountered if one combines an iterative development process with MDA-based development. With this thesis we want to obtain both operational knowledge and contribute to science. The operational knowledge is recorded in the form of this thesis and our case study. We provide an example of how MDA can be applied in an practical environment instead of a lab experiment. Our contribution to science is an overview of the issues that at least must be solved before MDA can be considered a mature technology.

This study has the following structure. The main part of our study is a case study. In the case study we will investigate how MDA can be applied in practice in an iterative software development process. The case study is based on a development project carried out by Getronics PinkRoccade. We will rebuild a part of the system they built in the Arend project. This system is a data centric web-application. In this study we rebuild the system using RUP as the development process and MDA to implement the system. The case study is documented in the form of plans and evaluations. The evaluations from the project are used in the second part of the study. In the second part we analyze the issues we have observed during our case study. For each of the issues we will provide a clear description of the problems, examples to illustrate the problems and a possible solution if there is a known solution.

1.5 CONTRIBUTIONS

The contributions can be summarized as follows:

Operational knowledge of MDA

An important part of this study is to provide knowledge and experience with MDA and iterative software development. We provide this knowledge in multiple forms. The most important form is this thesis. It describes our work and provides an evaluation of the project.

An overview of issues in the field of MDA

Because we carried out our project in an industrial setting we provide a different view of MDA. Researchers tend to focus on small problem areas and use laboratory experiments to verify hypothesis. During this case study we encounter practical problems that may not have been occurred in laboratory experiments.

More specific knowledge about MDA and iterative development

There is currently a lack of knowledge in applying MDA in an iterative fashion. With this case study we investigate what the implications are of combining MDA with an iterative development process. We use RUP as our iterative development process because it is an commonly accepted development process.

An example case of how MDA can be applied

This case study provides both the University of Twente and Getronics PinkRoccade with an example of how MDA can be applied in a practical situation. The project we selected for the case study is a project that is carried out by Getronics PinkRoccade for a customer. We took the requirements and illustrated how the same system can be built with MDA.

1.6 THESIS OUTLINE

The structure of this thesis is as follows. In chapter two the most important concepts and theoretical backgrounds of model driven engineering are laid out. The basics of the Rational Unified Process and the Open Unified Process are explained in chapter three. We used RUP as the software development method to develop our case. In chapter four the case study itself is described. In this chapter the architecture and design of the system are described in detail. All meta-models, models and transformations used to develop the system are part of it. In chapter five the development process is described. This chapter is like a log book of the project and contains the evaluations for each of the phases in the development process and a list of issues. In the last chapter, chapter six we write down our conclusions, recommendations and future work.

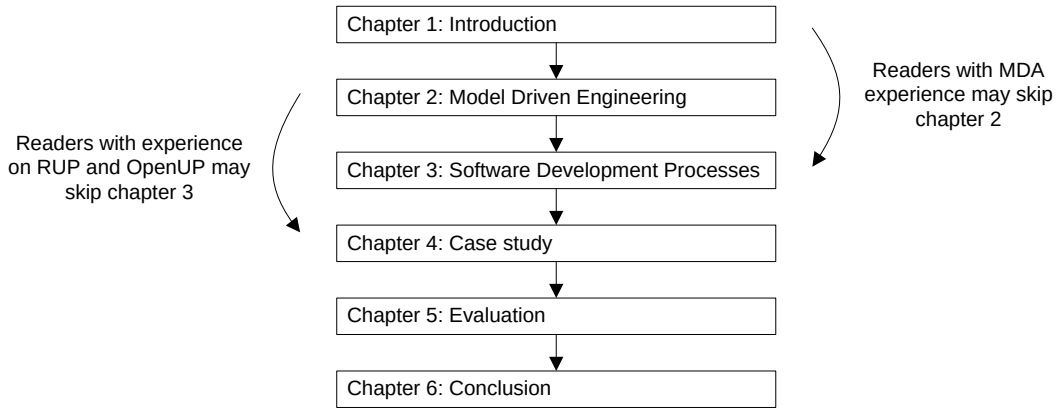


Figure 1: Thesis outline

Readers that have some experience with MDA may skip chapter two. We recommend reading chapter two because the definitions of important concepts are given and explained in this chapter. Chapter three is about iterative development processes. Readers with experience on RUP and OpenUP may skip this chapter (see Figure 1).

Chapter 2

2 MODEL DRIVEN ENGINEERING

This chapter gives an overview of the important concepts of model driven engineering. We explain these concepts and provide a theoretical background for model driven engineering. For most of the concepts we describe in this chapter there are no common accepted definitions. In this chapter we will define the concepts as they are used in this thesis.

2.1 MODELS AND MODELING

Models are used for various purposes. We can enumerate many different models: miniature models, weather models, mathematical models, economic models and of course UML models. The word model has its roots in Latin. The Latin word modulus means "a small measure" but there is no single definition of the word model. Actually, the dictionary (Mariam Webster) gives over 15 different meanings of the word model. The OMG defines a model with the following statement:

"A model of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language." (Object Management Group, 2003)

The definition from OMG describes a model as it is used in the context of MDA but it does not capture the fundamental concept of a model. For people with knowledge about modeling software systems, the definition from OMG describes how we often look at models. However, we need a more precise definition of a model. The following definition provides a more fundamental definition of a model:

"A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system." (Bézivin & Gerbé, 2001)

There is no general definition for the word model. In this thesis we use the definition from (Bézivin & Gerbé, 2001) because it defines a model in a broad sense. The models we are interested in are models that can be expressed in a modeling language.

Throughout this thesis we use the word modeling. With modeling we mean the act of creating or modifying models. We use the following definition for modeling:

"The process of construction or modification of a model." (American Institute of Aeronautics & Astronautics, 1998)

2.2 MODEL DRIVEN ENGINEERING

The term Model Driven Engineering (MDE) was introduced by Stuart Kent (Kent, 2002) and is used to refer to the general idea to use models as first class entities in software development. In MDE, models play a central role in the development process. Models provide both an abstract description of the system under development as they can represent the source code. In the literature one may find other terms such as Model Driven Software Development or Model Driven Development. All terms refer to the general idea of MDE. In this study we use the term MDE. The key aspect of MDE is that models no longer play a role as design tool or as reference. Instead, models form the basis of the development process. Different models can be used to describe a system and model transformations can be used to provide the semantics for these models.

One advantage of MDE is that it raises the level of abstraction in software development. By defining a system on a more abstract level, the complexity of designing systems can be reduced. The higher level of abstraction enables developers to concentrate on the important aspects of a system and forget about implementation details. Another advantage of MDE is that MDE forces developers to explicitly specify the system. If a model is used as an image to help designing the system, not all details have to be modeled. If a models is used to generate the code, the model should be complete.

2.3 META-MODELING

In MDE models are based on a meta-model. For example there is a UML meta-model that expresses UML class diagrams. The definition of meta-model used in this thesis comes from (Seidewitz, 2003):

"a meta-model is a model of models expressed in a given modeling language".

A meta-model is a model that defines the constructs which can be used to express models. A meta-model defines the structure of a model and the possible relations between the model elements. To be more precise, a meta-model defines the abstract syntax of a modeling language. Every model has a meta-model. Java programs have the Java grammar as their meta-model, and the Java grammar has a EBNF description as its meta-model (see Figure 2). Models and meta-models have a class-instance relationship.

Every model is an instance of a meta-model. If a meta-model is a model, the meta-model itself must have a meta-model. Conceptually, if every model has a meta-model we can visualize a stack of models where each layer is the meta-model of the layer below (see Figure 3). This stack of layers is called a meta-level architecture.

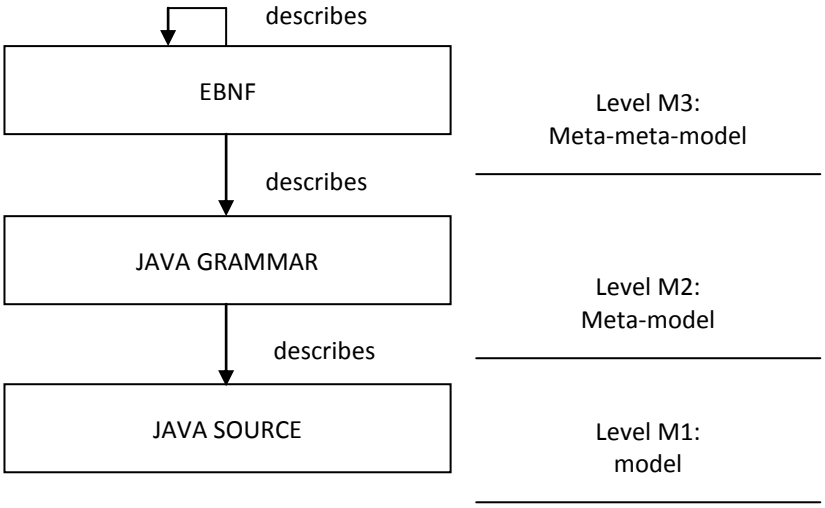


Figure 2: Meta-level example

In Figure 3, the levels are numbered. Level M0 refers to the domain that is being modeled. Level M1 refers to the model of the domain where level M2 is the meta-model and level M3 is the meta-meta-model. Every layer is an instance of layer M+1.

In theory the number of levels can be infinite, but a four level architecture is enough in practice. In this case, level M3 is self-descriptive. Level M3 can be specified using M3. In the example above (see Figure 2), EBNF is used to describe the abstract syntax (and in this case also the concrete syntax) of EBNF. The intuition behind the self-descriptive layer is quite elegant. In level M0 we have our domain which we model in M1. M2 is the language we use to give the abstract syntax to the modeling language and we have a language to describe the abstract syntax in M3. If this language is designed to describe the abstract syntax of a language it should be possible to describe itself (Kurtev, 2005).

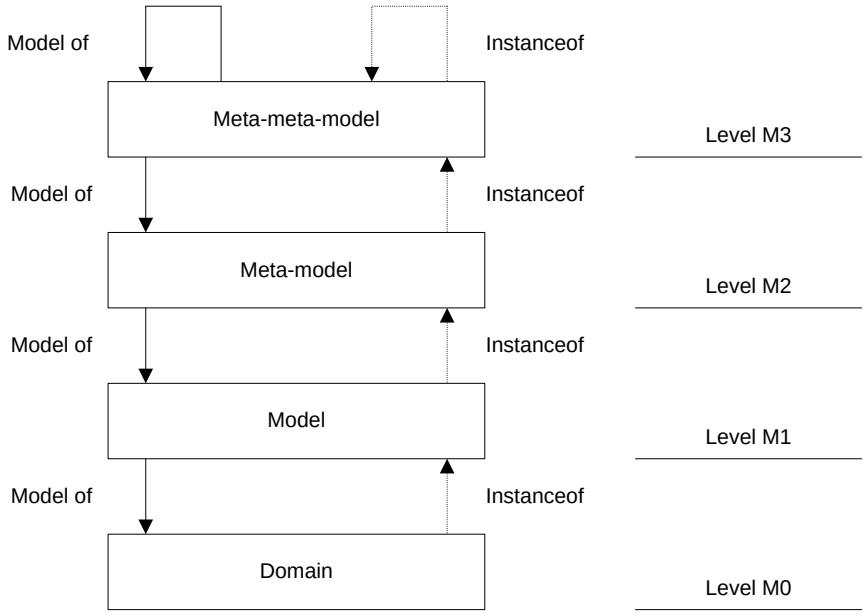


Figure 3: Meta-level architecture

The abstract syntax of a modeling language plays an important role in MDE. Tools, transformation languages and code generators can benefit from the fact that they are based on the same abstract syntax. The concrete syntax, the actual representation of a model, plays a less important role. It is considered a good practice to decouple the concrete syntax from the abstract syntax. This makes it possible to provide multiple concrete representations of the same model, for instance both a graphical and a textual representation. The concrete syntax is of course of great importance for the people who work with the models. A good concrete syntax is easier to work with and has a better readability and will improve the usage of the models (Object Management Group, 2003) .

2.4 MODEL TRANSFORMATION

Model transformations form a large part of MDE. The transformations are the main driver in the development process. Transformations can be used to convert one model into another model or to combine models. The term model transformation is a broad term. In this thesis we define a transformation as:

"the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language" (Kleppe, Warmer, & Bast, 2003).

This definition, however, states that we transform one model into one other model. As discussed in (Mens, Czarnecki, & Gorp, 2005) it should be possible to define transformations that transform multiple models into one model, multiple models into multiple models and one model into multiple models.

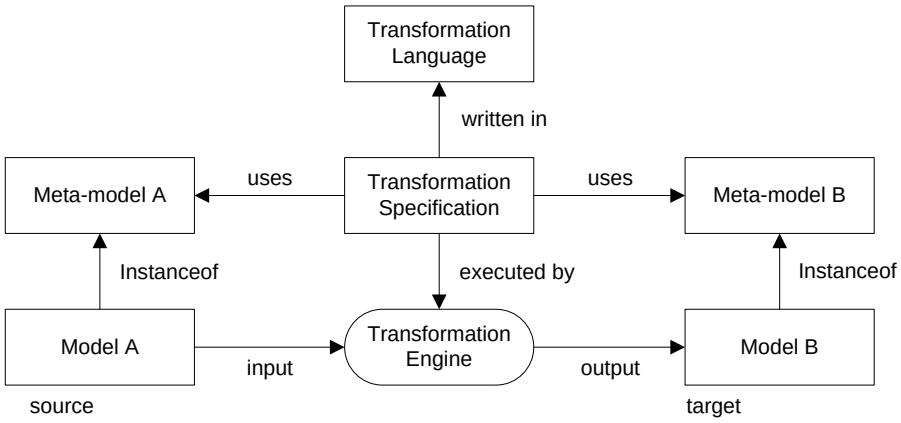


Figure 4: Meta-model transformation pattern (Kurtev & van den Berg, Building Adaptable and Reusable XML Applications with Model Transformations, 2005).

A model transformation can be described using the transformation pattern (see Figure 4). The transformation pattern shows how the transformation engine executes a transformation specification. This transformation specification is written in a transformation language and uses both the meta-model of the target model and the meta-model of the source model. The transformation specification specifies how elements of meta-model A can be transformed into elements of meta-model B. Using this specification the transformation engine can transform the input model (Model A) into the target model (Model B).

In the literature different types of transformations are identified (Mens, Czarnecki, & Gorp, 2005) (Czarnecki & Helsen, 2003). These includes transformations between models with the same meta-model called *endogenous* transformations and transformations with different meta-models called *exogenous* transformations. In the literature one can find a difference between *model to model* (m2m) and *model to text*(m2t) transformations. There is a small difference between those types of transformations. The difference can be identified when we look to the nature of the transformation. In a model to model transformation a mapping between the meta-models is defined while transformation templates are used for m2t. The output produced by m2t transformations can be seen as a model with an implicit meta-model while the output of the m2m transformation has an explicit meta-model. Most m2t transformations are exogenous transformations. In this case the generated text is on a more concrete abstraction level than the source model. However, endogenous m2t transformations also exist. Generating Java source code from a Java abstract syntax tree is an example of this. In this thesis we will make the distinction between m2t and m2m transformations for the sake of readability but conceptually difference is minimal.

Another distinction between types of transformations is the distinction between *horizontal* and *vertical* transformations. A horizontal transformation is a transformation where the input and output models have the same level of abstraction. An example of a horizontal transformation can be refactoring. With refactoring we only change the structure of the code. Vertical transformations transform a model to a model of a higher or lower abstraction level. An example of a vertical transformation is a refinement.

	Horizontal	**Vertical**
endogenous	*Refactoring*	*Formal refinement*
exogenous	*Language migration*	*Code generation*

Table 1: Dimensions of model transformations[1]

Transformation languages come in many forms. Just like programming languages can be classified, we can classify transformation languages based on how they allow the programmer to express the transformation. The two most distinct types are *operational* and *declarative* languages. Declarative languages focus on *what* must be transformed into what. Declarative languages seem to be the most promising and there is a solid formal basis for these languages. It is easier to implement bi-directionality. These languages are easier to use because the engine that executes the transformation determines the execution ordering and model traversal. An operational (also called imperative) language focuses on *how* the transformation must be applied, giving the programmer a tool to precisely define the execution order and fine grained control over the transformation. As a result of this, transformations expressed in an operational language are often larger and more complex. Other types of languages could also be interesting to use, like graph transformations and logical languages (Mens, Czarnecki, & Gorp, 2005). A special type of language is the hybrid language. Hybrid languages are a mix of different types, usually with imperative and declarative constructs.

Examples of transformation languages are Stratego (Visser, 2004), KerMeta (Chauvel & Fleurey, 2007), Tefkat (Lawley), ATL (Allilaire, Bézivin, Jouault, & Kurtev, 2006) and QVT (Object Management Group, 2007). Stratego is a transformation language that is used in combination with XT. This bundle is called Stratego/XT. Stratego is a transformation language based on term rewriting. KerMeta is an abbreviation of “Kernel Metamodeling” and is a combination of a modeling and transformation

[1] It is not my intention to provide a complete taxonomy of transformation languages in this thesis. For further reading see (Mens, Czarnecki, & Gorp, 2005) (Czarnecki & Helsen, 2003).

language. The language can be used to create models, to put restrictions on models and to transform models. Tefkat, ATL and QVT are all implementations of OMG's MOF Query/Views/Transformations RFP. QVT was finally adopted as the standard by the OMG.

A special type of transformation is currently gaining the interest from the research community: model composition. When we want to combine multiple source models into one target model this is called model composition. Model composition is a special form of transformation because it does not transform a model. However, it combines multiple models into one (Kurtev & Didonet Del Fabro, 2006) (Pastor, 2006).

2.5 DOMAIN SPECIFIC LANGUAGES

Computer scientists have always been working on languages. Even before the first computers were build, scientists were thinking of a way to program the hypothetical machines[2]. Currently a large variety of programming languages exist. Some of them such as Java, C# and C++ are more generic; some of them such as Cobol and Fortran are more specific. This section gives an introduction to domain specific languages. With domain specific languages we mean languages that are built to solve problems in a small problem domain. Their counterparts, generic programming languages, are built to solve problems in a larger problem domain. Of course, the terms generic and specific are relative. One language is more specific than another language. DSLs are not a new concept in computer science. Many engineers already use DSLs in their daily work without referring to them as a DSL. SQL is a nice example of such a DSL. It is a limited language which can only be used to query databases. Therefore, it is impossible to use SQL to create a socket connection and transfer a file. Since SQL has a specific domain it can provide a simple syntax which enables the programmer to specify complex queries with low overhead. Another example of a DSL is a configuration file or an EBNF grammar (Kurtev, Bézivin, Jouault, & Valduriez, 2006).

In this thesis we will use the following definition for DSLs:

"A domain-specific language is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain" (Deursen, Klint, & Visser, 2000).

DSLs can be implemented in a number of ways. One could use standard language engineering tools like ANTLR to build an interpreter or compiler and generate a large part of the implementation. The advantage of this approach is that you can actually implement the language without having to do any concessions on the language. With complete power over the language you can implement a complex type system, type checker, error checking and optimizations. The drawbacks on the other hand are the costs of the implementation and less reusability. Another approach could be to implement the DSL using special language constructs. Some general purpose programming languages like Ruby allow the programmer to implement a DSL on top of the base language. When you build a DSL based on an existing base language, one could use macros which are fed to a preprocessor. The preprocessor then translates the macros into constructs of the base language. One advantage of this approach is that the compiler is able to detect errors but the disadvantage is that error messages are given at the level of the base language. Sometimes even an existing compiler or interpreter can be extended to support a DSL. The Tcl interpreter is an example of this approach (Mernik, Heering, & Sloane, 2005) (Deursen, Klint, & Visser, 2000).

[2] Charles Babbage designed a mechanical computer called "The Analytical Engine" in 1837, but the machine was never build. Ada Lovelance came up with an algorithm to compute Bernoulli numbers on the mechanical computer and she is considered the first computer programmer. The programming language Ada is later named after her.

A modern approach is to use aspect oriented programming (Elrad, Filman, & Bader, 2001) to weave the DSL code into the base code. An example of this approach can be found in (Oever & Vos, 2007) where a DSL is described to specify user profiles for applications. The profiles are then woven into the code by using an aspect oriented approach.

2.6 MODEL DRIVEN ARCHITECTURE

The Object Management Group developed MDA as an approach for MDE based on industry standards and with the main focus on platform independence. According to OMG, MDA provides an approach for specifying the system and its behavior independently of the platform that supports it and for specifying the platform itself (Object Management Group, 2003).

OMG describes a system as:

"We present the MDA concepts in terms of some existing or planned system. That system may include anything: a program, a single computer system, some combination of parts of different systems, a federation of systems, each under separate control, people, an enterprise, a federation of enterprises…"

OMG describes a platform as:

"A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented"

The main purpose of MDA is to provide platform independence and the different models used in MDA reflect this thought. In MDA there are three types of models: the Computational Independent Model (CIM), the Platform Independent Model (PIM) and the Platform Specific Model (PSM).

The CIM is the model that describes the systems requirements without any references to a particular technology or technique and it does not show how the system is implemented. The CIM shows the systems in its environment and helps presenting what the system should do. The CIM is not widely used because of the vague definition of what it actually should be. The model is also not directly used to create a PIM. However, there must be a clear relation between CIM and PIM.

The PIM is an important part of MDA. It describes the system without referring to the platform and thus gives a platform independent representation of a system. This model is the first step to create a system. The PIM can be transformed into a PSM using model transformations.

The PSM models provide information about how the system is implemented and its relation to the platform. A PSM contains the information from the PIM with specific details about the target platform. For example, a PIM may contain a concept with properties and one of the properties is whether it should be persistent or not. The PIM does not contain information about how the concept should be persisted but when the PIM is transformed to the PSM for a J2EE platform, the concept will be mapped to a Java Bean with the proper annotation for the persistence API[3].

[3] As earlier stated, code can be regarded as a model. In the case of MDA code can be considered a very detailed PSM. However, code has a very low level of abstraction and therefore it is preferred to make the distinction between a PSM and the code.

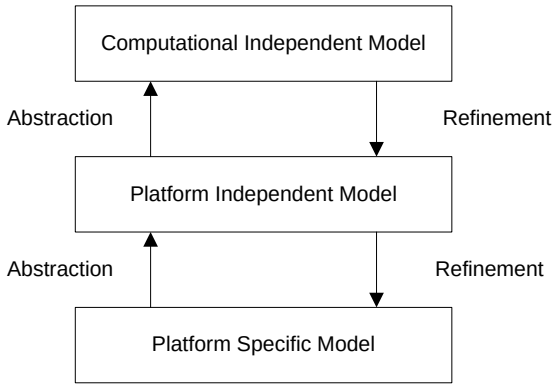


Figure 5: MDA layered model structure

Figure 5 shows the types of models and the relations between them. Each level represents a type of models, not one model. The refinement arrows indicate that a CIM can be refined to a PIM and that a PIM can be refined to a PSM. This refinement step is not necessarily one transformation step but it can be specified in multiple transformation steps. The abstraction arrows indicate that a PIM is more abstract than the PSM and the CIM is more abstract than the PIM.

2.7 META OBJECT FACILITY

Modeling in MDA is based on standards like UML 2.0 and MOF. MDA uses a four level architecture (see Figure 3) and the meta-meta-model is the Meta Object Facility (MOF). For historical reasons the original UML model had no explicit meta-model. MOF was later on defined as the meta-model of UML. MOF was directly derived from the UML standard and reflects the structure of UML. The origin of MOF can be seen in the MOF concrete syntax, it uses the UML notation. The syntax is similar to the UML class diagram notation. MOF forms the basis for OMG's MDA. Figure 6 shows a simplified MOF structure that contains the basic elements to describe modeling languages like UML 2.0 (Object Management Group, 2006).

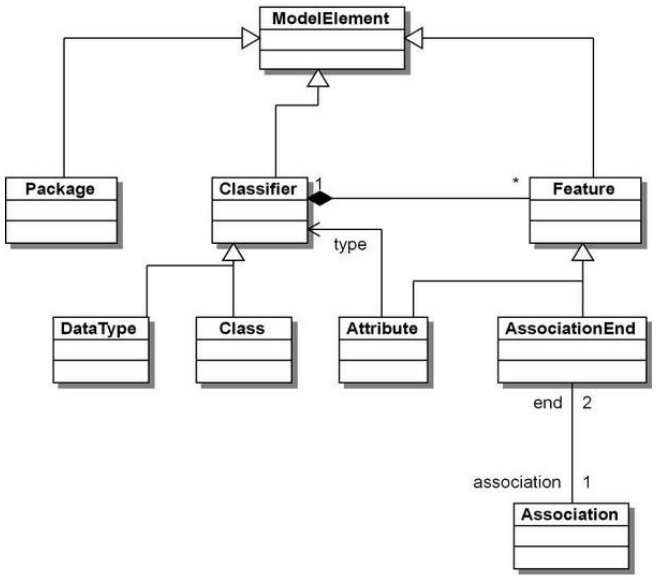


Figure 6: Simplified MOF structure[4]

[4] Copyright Detlef Burkhardt

A common mistake is to think that UML is the only meta-model supported by MDA but that is not the case. UML is supported because it is now formally defined in MOF. All languages defined in MOF can be used in the MDA approach and tooling. Examples of other languages based on MOF are SPEM (Object Management Group, 2005), CWM (Object Management Group, 2003) and QVT (Object Management Group, 2007). However, for the UML models advanced editors are available which is not always the case for other meta-models. Generic editors, generators, repositories and tools can be implemented because MDA is based on MOF.

At the time of writing this thesis a number of MOF implementations exist. Probably the most well known implementation is the EMF (Eclipse Modeling Framework) with the eCore meta-meta-model. eCore is not a full implementation of MOF. Only the most important parts of it were implemented. This smaller subset is enough for the major part of the meta-modeling. This subset even made it into the latest MOF standard. It is called Essential MOF (EMOF). OMG positions EMOF as the meta-meta-model to define simple models while supporting extension. For more sophisticated models, the CMOF (Complete MOF) (Object Management Group, 2006) can be used.

2.8 UNIFIED MODELING LANGUAGE

The Unified Modeling Language (UML) is the unification of three modeling languages. UML was originally created by Grady Booch, Jim Rumbaugh, and Ivar Jacobson. UML was created to provide a standard visual modeling language for modeling object oriented software systems. In 1996 the OMG issued a request for proposal for a standard object oriented modeling language. Booch, Rumbaugh and Jacobson began preparing a proposal based on UML. In 1997 the UML 1.0 standard was accepted by the OMG (Rumbauch, Jacobson, & Booch, 1999).

UML offers five views of a system: logical, process, physical, development and use-case view. These views together are called the “4+1” view (Kruchten, 1995). The logical view is the object oriented model. In UML this can be described using class diagrams. The process view captures synchronization and concurrency aspects. Activity diagrams are often used to describe this view. The physical view describes how the software can be mapped to the hardware. It shows the distributed aspects. In UML, deployment diagrams can be used to document this view. The development view describes the static organization of the software. In UML packages can be used to organize the static structure of software. The “plus one” view is the scenario view. This view is represented with use-cases in UML.

The UML 2.0 standard is one of the foundations of MDA but UML 2.0 is a general purpose modeling language. In some cases more specific models are needed and MDA provides two ways to create a more specific modeling language. The first method is to create a modeling language based on the MOF. The second method is to use UML Profiles. UML Profiles were invented to make specific flavors of UML, tailored for one purpose. In the early specifications it consisted of stereotypes and tagged values. These were no more than textual annotations. In the UML 2.0 standard the stereotypes and tagged values still exist but also constraints are added. In UML profiles one can specify a flavor of UML. However, the semantics of UML cannot be changed. It is possible to add semantics that are left unspecified in the UML specification. Using the constraints it is also possible to limit the ways to use the meta-model. UML Profiles also add some syntactic sugar like custom graphical representations. This can be used to show a computer in a network as a computer icon instead of a rectangular box (Object Management Group, 2007) (Fuentes-Fernández & Vallecillo-Moreno, 2004).

Nowadays many UML profiles exist. An example of an UML profile is a profile for aspect oriented programming (Aldawud, Elrad, & Bader, 2003) or a profile for business modeling (Rational, 2004) .

2.9 OBJECT CONSTRAINT LANGUAGE

In general, a UML model is not refined enough to specify all details of a software system. Some aspects, like pre and post conditions, queries and conditions are hard to specify. In early UML standards the only way to specify these details was to use textual notes in the model. But these notes are not machine readable and can lead to ambiguities. The Object Constraint Language (OCL) was developed to fill that gap (Object Management Group, 2003).

OCL is a small language and not a real programming language. OCL is more like a specification language and control flow cannot be specified. In OCL only expressions without side effects can be described. These expressions can be evaluated over a model and a value is returned but OCL expressions cannot change the model. OCL is a typed language and has a set of predefined types. OCL can be used with any MOF based model and every classifier can be used as a type.

```
1  Class.allInstances()->collect(e | e.name)
```

Code section 1: A sample of OCL

Code section 1 shows a simple OCL expression. This expression can be evaluated over a UML model and will then return a bag with all the names of the classes in this model. This expression calls the "allInstances()" operations on the type "Class". This operation returns a bag with all instances of the type "Class" in a particular model. The "collect" operation collects the results of the expressions for each element in the returned bag of classes. The expression "e.name" returns the name of a class.

2.10 QUERY VIEW TRANSFORMATION

MDA prescribes the use of model transformations between different models. A PIM can be transformed into a PSM using a model transformation. Such transformations between models are defined in a transformation language. MDA uses the Query/View/Transformation (QVT) language (Object Management Group, 2007). This transformation language acts on models. It defines how a model M^A with meta-model M^{MA} is transformed into a model M^B with meta-model M^{MB}. Model M^A and M^B may share the same meta-model but this is not necessary. In MDA, all meta-models and languages that operate on them must be based on MOF. QVT is also based on MOF.

As the name suggest this language has multiple purposes. The language can be used to specify queries, views and transformations. In (Gardner, Griffin, Koehler, & Hauser, 2003) the following definitions are given for the terms view, query and transformation:

view: "A view is a model which is completely derived from another model".
query: "A query is an expression that is evaluated over a model".
transformation: "A transformation generates a target model from a source model".

QVT is not a single language. It actually implements OCL as the query language and has two transformation languages. These language can also be used to specify views, as a view is the result of a transformation. The two languages are: QVT Relations and QVT Operational Mappings. A third language forms the foundation for Relations and Operational Mappings: QVT Core. In general QVT is a hybrid transformation language with a declarative and imperative nature. The declarative part is split in Relations and Core. The imperative part is Operational Mappings. Relations is a language in which the relationships between MOF meta-models can be declaratively specified. The relations language provides a graphical and textual concrete syntax and can be transformed to Core using model transformations. Relations supports object pattern matching and implicitly creates trace classes.

Operational Mappings is an imperative language that extends both Relations and Core. It provides control flow elements and a textual concrete syntax.

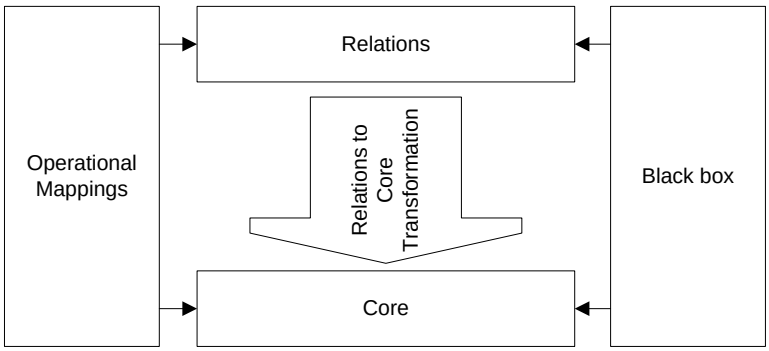


Figure 7: QVT layered architecture

Figure 7 shows the layered structure of the three QVT languages. Operational Mappings is based on both Relations and Core, Relations can be transformed to Core. The black box in the figure is a mechanism to invoke transformations specified in other languages. For instance, a complex algorithm can be specified in a general purpose programming language that has a MOF binding.

In general, QVT can only be used for model to model transformations. Model to text transformations are not included in the specification.

```
1  --generate an overview page for this node (using the nodename)
2  mapping createOverviewPage( in element : diagram::Node) : applicationmodel::Page {
3       name := 'Overview of '+element.name+'s';
4  }
```

Code section 2: A sample of QVT Operational Mappings

Code section 2 shows a mapping operation in QVT Operational Mappings. A mapping operation describes how an element from the source model is mapped to an element from the target model. In this example a Node is mapped to a Page and the operation determines how the name of the node is mapped to the name of the page.

2.11 CONCLUSION

In this chapter, we presented the backgrounds and theory behind MDE. The definitions for the concepts used throughout this thesis were defined and explained. The terms model, meta-model, transformation and domain specific language were defined. Models can have a abstract and a concrete syntax. The concrete syntax can be textual or graphical. We explained the meta-level architecture and the fact that level M3 is self descriptive. We also discussed MDA as a form of MDE and the technological foundation of MDA. MDA is based on MOF, UML, OCL and QVT. Each of these techniques is described in a separate section.

Chapter 3

3 SOFTWARE DEVELOPMENT PROCESSES

In this chapter we will describe the Rational Unified Process as an example of an iterative software development process. As we are interested in the problems that occur when MDA is applied with an iterative development process we need to have a common understanding of such a process. In our case study we will apply the Rational Unified Process because it is currently a de facto standard (Kruchten, 2000).

3.1 SOFTWARE LIFE CYCLE

One of the most fundamental concepts in software engineering is the software life cycle (Glenn Brookshear, 2000). Figure 8 shows the software life cycle. This figure illustrates how the software cycles trough its life from being produced, used and modified. Like other manufactured products software needs modification once the product is in use. In contrast with other products, part of the software do not wear out. Software requires changes because the it contains errors, the environment changes or the demands of the users change.

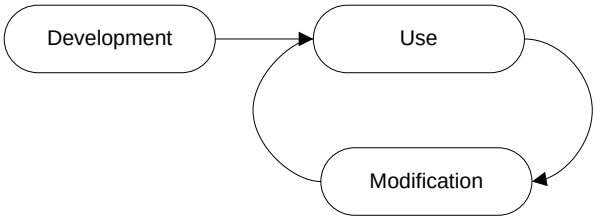


Figure 8: Software life cycle

The phases of the software lifecycle that we are interested in are the development phase and the modification phase. Software development processes like the waterfall model describe the structure of the development phase of a software product. Many different models exist to describe this phase. For instance the spiral model, the V-model, the incremental model and the prototyping model (Pfleeger & Atlee, 2005). In this study we describe the waterfall model because it is argued that MDA follows a waterfall process (Wegener, 2002). We the describe the iterative process because we use an iterative development process in our case study.

3.2 WATERFALL MODEL

The software development process can have many forms. The ordering in which different activities are executed determine what type of process we have. The traditional waterfall model has six phases of development. Only if a phase is completed one may start working on the next phase. The model was first described by (Royce, 1987).

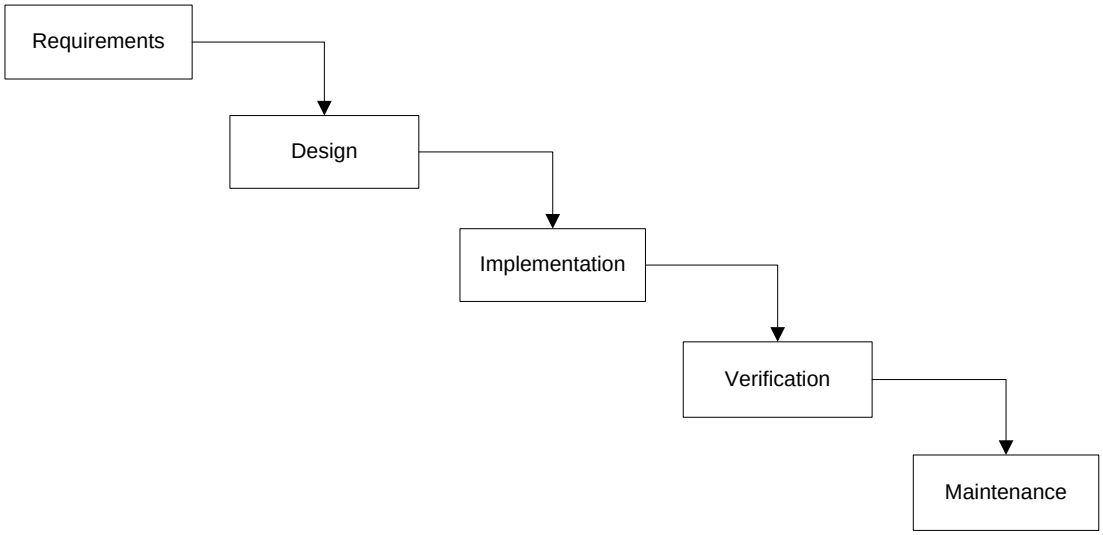


Figure 9: Simplified schema of the waterfall model

The above figure (Figure 9) illustrates why it is called the waterfall model. First the requirements for a system are gathered and documented. Once this is done the requirements are fixed and the next step can be done: the design. Once the design of the system is completely finished one can start implementing it. After the implementation phase the test phase starts. Problems detected in this phase can be fixed and once the system is considered finished the maintenance phase starts.

This software development is considered a risky model because it is impossible to go back to an earlier phase. Once the system is implemented completely testing may begin. If major problems are detected in this phase, maybe design flaws, the whole system is already build. Changing the design at that stage of development is very expensive (Royce, 1987)

3.3 ITERATIVE MODEL

The iterative software development model is a model that was developed as a response to the harmful waterfall models (Kruchten, The Rational Unified Process: An Introduction, 2000). One of the problems of the waterfall model is that testing at the end of the development cycle may reveal problems that could have been solved in earlier stages of the project.

In an iterative development process the development of the system is small steps called iterations. In every iteration a part of the system is designed, implemented and tested. Due to the small steps, problems can be detected in early stages and developers can learn from previous iterations. It is not necessary that every iteration contains a part of requirements, design, implementation and verification. In the early stages of a project more time can be spend on requirements and in later stages more time can be spend on verification (Basili & Larman, 2003).

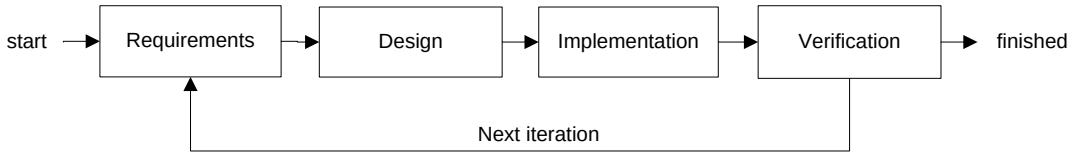


Figure 10:Schema of the Iterative model

There is a difference between incremental software development and iterative development. Both models divide the project in smaller pieces of work but in an incremental model the software system can only be extended in each iteration. In an iterative development model it is possible to build a piece in the first iteration and discard the work in the second iteration.

3.4 RATIONAL UNIFIED PROCESS

The Rational Unified Process (RUP) is an iterative software development process. The goal of RUP is to ensure the production of high quality software that meets the needs of its end users, on schedule and within budget. RUP provides a very systematic approach that defines roles and tasks for the organization of the project. RUP has been used for both small and large teams and long and short projects (Kruchten, The Rational Unified Process: An Introduction, 2000) (Rational Corporation, 1998).

3.4.1 BACKGROUND

The Rational Unified Process has a long history and was originally developed by Rational Corporation in the 1980's and 1990's. In 1995 Rational Corporation bought a Swedish company called Objectory AB. Their process, called Objectory process is combined with the knowledge of Rational at that time and Rational releases the Rational Objectory Process (ROP) in 1996. In 1998 it is renamed Rational Unified Process. The architect of RUP was Philippe Kruchten. In 2003 IBM acquired Rational and became the division IBM Rational (Amber, 2007).

The authors of RUP found that many of the software projects in the 80's and 90's were failing. They tried to find the main cause for the problems and tried to diagnose what went wrong. They diagnosed different characteristics of software projects and came up with a number of causes for software project failures:

- Ad hoc requirements management
- Ambiguous and imprecise communication
- Brittle architecture
- Overwhelming complexity
- Undetected inconsistencies in requirements, designs, and implementations
- Insufficient testing
- Subjective assessment of project status
- Failure to attack risks
- Uncontrolled change propagation
- Insufficient automation

Each failed software project that was investigated had failed because one or more of these failures had occurred. Rational Corporation used this knowledge to come up with a structured system of best practices to cope with the listed problems. These best practices cannot be easily quantified, but are

used in RUP because it is observed that these are commonly used in the industry by successful organizations. RUP focuses on the following six best practices (Rational Corporation, 1998):

- *Develop software iteratively*: The software systems that are developed today are complex systems. It is no longer possible to sequentially design, implement and test these systems in the end. In an iterative development process, understanding of the system is developed in small steps of refinement called iterations. In RUP, each of the iterations delivers a working product. The iterative nature of the project helps project management to identify and mitigate risks in early stages of the project. Because each iteration ends with a working product, it is also easier for customers to get involved in the project and to correct misunderstandings in early stages of the project.

- *Manage requirements*: RUP describes precisely how to elicit, organize and document required functionality and constraints. RUP describes functionality in terms of use-cases and scenario's. This will help communication with the business and is considered a good practice to capture functional requirements.

- *Use component-based architectures*: In the early phases of development, RUP focuses on developing a baseline architecture that provides a solid base to develop the entire system. RUP helps designers to create a flexible, reusable and easy to understand architecture. RUP also provides the means to document the architecture trough UML models.

- *Visually model software*: An important aspect of RUP is the use of UML throughout the design of a system. It prescribes different views for different stakeholder documenting the structures of the system in diagrams. The UML standard was originally developed by Rational Corporation.

- *Verify software quality*: Today's ever increasing complex software systems play an important role in our everyday lives. Software quality is becoming an important aspect and RUP helps developer creating high quality software. Quality assessments are part of the process, in all activities and phases.

- *Control changes to software*: Managing changes in the software is part of the process. RUP describes how changes must be controlled, checked and monitored. It provides guidelines on how to setup the environments for developers so they are only allowed to change their own code.

3.4.2 PHASES

The overall architecture of RUP is illustrated in Figure 11. The horizontal axis represents time and shows the phases of a RUP project. Every phase in a project contains a number of iterations. The vertical axis represents the different disciplines grouped logically and shows the static aspects of the process. Activities from different disciplines are overlapping and the shaded areas indicates the effort for that particular discipline at that time.

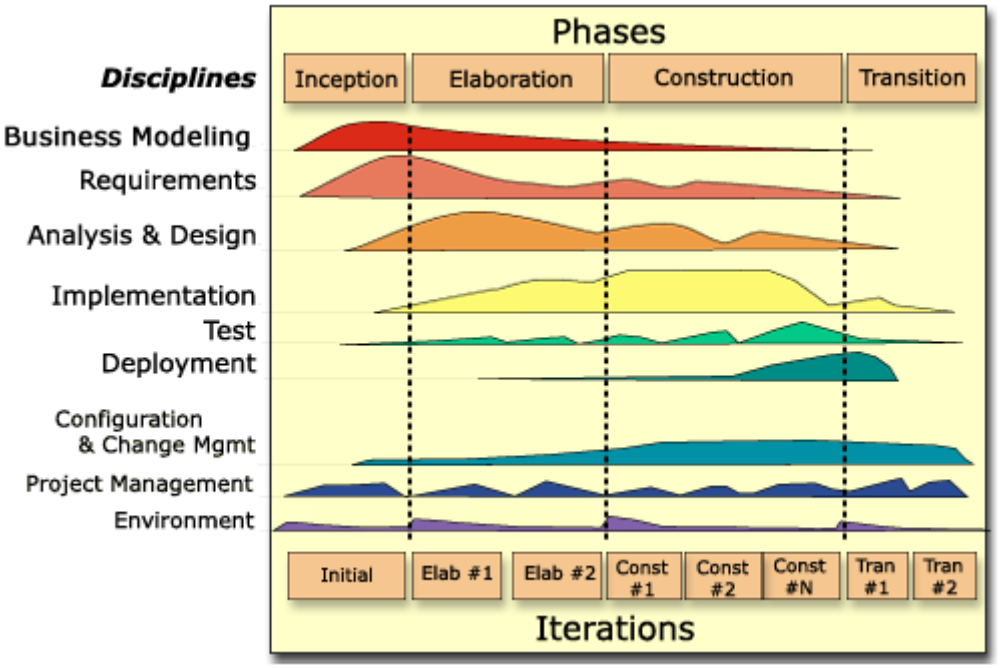


Figure 11: Phases and iterations of RUP[5]

From a project management perspective, RUP consists of four sequential phases: the inception, elaboration, construction and transition phase. Each of these phases can contain multiple iterations as shown in Figure 11 and ends with a milestone (dashed line). At the end of a phase, there is an assessment to verify if the goals for that phase are met. If the evaluation shows all goals are met, the project is allowed to proceed with the next phase.

- *Inception*: The project starts with the inception phase. The goal for this phase is to reach agreement among different stakeholders of the project. Before the project can start, some business risks should be addressed and the initial requirements should be known. The focus of this phase is finding out if the project is worth taking the risk and whether the project is feasible. The assessment at the end of this phase is to evaluate if the project can continue.

- *Elaboration*: In the elaboration phase, the baseline architecture of the system must be built. This baseline will be used in the next phase to build the major part of the system. In this phase, an executable system will be built that contains the most important requirements and shows the viability of the architecture. The exercise of building this baseline architecture should identify the risks in the project.

- *Construction*: During the construction phase the remaining part of the requirements are clarified and the baselined architecture is completed. The construction phase can be considered the manufacturing phase of the system. In the previous phases, the focus was the intellectual challenge, but in the construction phase the focus is managing resources and developing the major part of the system. At the end of the construction phase, the system is tested and is ready for acceptance testing.

- *Transition*: The goal of the transition phase is prepare the system for its end users. It includes testing in preparation for release. Only minor adjustments and tweaks are done to the

[5] Copyright IBM

system at this stage. This phase also contains configuration, installation and addressing usability issues. The assessment at the end of the phase is to decide if the system meets the objectives.

3.4.3 ROLES

In RUP there are 24 roles defined. There are two or more roles for each of the nine disciplines. Essentially, there is division between two types of roles. A role for a person that focuses on breadth and a role for a person that focuses on depth. The following roles are defined:

- *Business Process Architect:* Discovers all business use-cases.
- *Business Designer:* Details a single set of business use-cases.
- *System Analyst:* Discovers all requirement use-cases.
- *Requirements Specifier:* Details a single set of requirements.
- *Software Architect:* Decides on technologies for the whole solution.
- *Designer:* Details the analysis and design of a single set of use-cases.
- *Integrator:* Owns the build plan that shows what classes will integrate with one another.
- *Implementer:* Codes a single set of classes or a single set of class operations
- *Test Manager:* Ensures that testing is complete and conducted for the right motivators.
- *Test Analyst:* Selects what to test based on the motivators.
- *Test Designer:* Decides what tests should be automated and creates automations.
- *Test Designer:* Implements automated portions of the test design for the iteration.
- *Tester:* Runs a specific test.
- *Deployment Manager:* Oversees deployment for all deployment units.
- *Tech writer, course developer, graphic artist:* Create detailed materials to ensure successful deployment.
- *Project Manager:* Creates a business case and a course-grained plan.
- *Project Manager:* Plans, tracks and manages risk for a single iteration.
- *Process Engineer:* Owns the process for the project.
- *Tool Specialist:* Creates guidelines for using a specific tool.
- *Configuration Manager:* Sets up the CM environment, policies and plan.
- *Change Control Manager:* Establishes a change control process
- *Configuration Manager:* Creates a deployment unit, reports on configuration status, etc.
- *Change Control Manager:* Reviews and manages change requests.

3.4.4 RUP AND MDA

The application of MDA in a RUP project requires some changes in the software development process. New artifacts need to be introduced to the development process as RUP does not consider model transformations. Also, new roles can be introduced. An IBM article describes a number of changes in RUP, based on their experience with MDA and RUP.

According to (Brown & Conallen, 2005), one new role can be added to the development process. This new role is a specialization of the architect called the MDA architect. This person is responsible for designing the main meta-models, transformations and mapping documents.

Other roles remain unchanged as the nature of the work remains the same. There is, however, a change in perspective. The level of abstraction for many roles will shift from concrete to more abstract. An example of this is that programmers in an MDA project are creating detailed models and

transformations instead of writing code based on UML diagrams provided by the designers. The focus of designers is more in the business level then in technical level.

The article also describes some changes in the phases. The phase that changes most is the elaboration phase. In this phase, a large part of the MDA work needs to be done. The meta-models and basic transformations need to built to see if the approach is feasible. Work in the construction phase will shift to modeling and writing transformations instead of working with source code.

One of the best practices of RUP is to use a component based architecture. MDA augments the use of component based architectures by providing the means to automate a large part of the work on integrating the components.

3.5 OPEN UNIFIED PROCESS

This section describes the Open Unified Process (Balduino, 2007) and the Model Driven Development plug-in for this development process. The Open Unified Process (OpenUP) is an open source version of the Rational Unified Process. It has been developed as a part of the Eclipse Process Framework project. This project provides an open and extensible framework for developing and maintaining processes and tools.

3.5.1 DESCRIPTION

The OpenUP is a family of software development processes built on top of the OpenUP/Basic. The OpenUP/Basic is an iterative software development process that claims to be minimal, extensible and complete. With a minimal process, the developers mean a process that contains only fundamental content. Extensible means a process that can be used as a basis for other processes and complete indicates that the process covers all aspects of software development (Lyons, 2007).

The OpenUP/Basic is a development processes that follows an agile philosophy (Ambler, 2007). The development process focuses on collaboration and delivering working products rather than focusing on formality.

OpenUP focuses on the following four principles:

- Collaborate to align interests and share understanding.
- Balance competing priorities to maximize stakeholder value.
- Focus on the architecture early to minimize risks and organize development.
- Evolve to continuously obtain feedback and improve

Agile software development is an incremental and iterative approach which is performed in a highly collaborative way using self organizing teams with low overhead and the focus on high quality software, delivered on time and cost effective (Ambler, 2007). Agile software development is based on the Agile manifesto (Beck, et al., 2001). This manifesto describes the core values of the agile philosophy: Individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, responding to change over following a plan.

Each of the principles of OpenUP is related to a statement in the agile manifesto. OpenUP is lightweight and an agile process but an agile process is more than lightweight. The OpenUP combines agile aspects like the focus on team collaboration and less formality then RUP. The OpenUP has the

characteristic of a light weight RUP; it is based on use-cases, iterations, risk management and an architectural centered approach.

OpenUP addresses the organization of work at three levels: personal, team and stakeholder levels. At the personal level, the work is organized in micro increments. Small units of work that span several hours or days. Team members share their progress at all three levels to monitor project progress.

3.5.2 PHASES

The division in phases and iterations reflects the origins of OpenUP. The project contains the same four phases as RUP: inception, elaboration, construction and transition. Each phase can contain multiple iterations and each iteration contains micro increments. At the end of each iteration, the project team delivers a testable demo or shippable build. This helps the team focus on it goals and offers the stakeholders a predictable lifecycle.

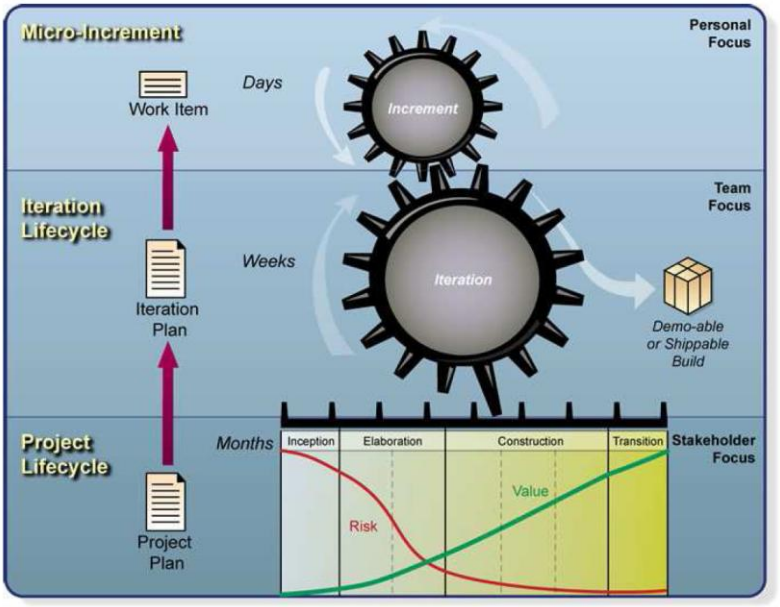


Figure 12: OpenUP Structure (Balduino, 2007)

3.5.3 ROLES

In the OpenUP, the following roles are defined for team members (Balduino, 2007):

- *Stakeholder:* A person or interest group whose needs must be satisfied by the project. Can be represented by anyone who is or will be materially influenced by the project.
- *Analyst:* Represents customer and end-user concerns by gathering input from the stakeholders to understand the problem to be solved.
- *Architect:* Responsible for designing the software architecture and for making key technical decisions.
- *Developer:* Responsible for designing, implementing, testing and integrating parts of the solution to fit in the architecture.

- *Tester:* Responsible for all the core testing activities. This includes identifying, implementing, and conducting tests as well as logging and analyzing the results.

- *Project manager:* Leads the planning of the project and handles communication with stakeholders. Keeps the team focused on the project goals.

3.5.4 OPENUP AND MDA

The OpenUP/Basic has been developed to form the basis for other development processes. In 2006, the development of a Model Driven Plug-in for the OpenUP started based on OpenUP/Basic. This plug-in provides new process elements that are specific for a model driven development project.

The OpenUP/MDD plug-in redefines a number of roles and provides a number of new roles that extend the existing roles of OpenUP:

- *Analyst:* The role of the analyst changes. The analyst should be aware of the fact that problems should be analyzed at model level.

- *Application designer:* Designs the transformations from PSM to code.

- *Business expert:* Designs profiles for specific business domains

- *Domain expert:* Has a detailed understanding of certain domains.

- *Language engineer:* Is an expert in modeling languages.

- *Platform expert:* Is responsible for defining platforms. These platform specifications can be used to create the models at PSM level.

- *Requirements specifier:* Specifies the requirements.

- *Test designers:* The person responsible for defining the test approach.

- *Transformation specifier:* Responsible for the specification of the PIM to PSM model transformations.

The OpenUP/MDD plug-in is created as part of the Eclipse project and can be found here: (Eclipse Process Framework OpenUP/MDD, 2006).

3.6 CUSTOMIZATION OF RUP AND OPENUP

Both RUP and OpenUP are complete software development processes that contain detailed descriptions of each step in the process. However, not every company uses exactly the same process and therefore these processes must be adaptable to their environment.

RUP provides a development kit which contains guidelines, templates and even a tool to adapt the process. Using the guidelines, templates and tools a complete documented and customized version of RUP can be generated and maintained (Rational Corporation, 1998). OpenUP is based on the Eclipse Process Framework (EPF). This framework is created to support the development of customized

software development processes. The EPF provides the EPF Composer as a tool to define and adapt software development processes (Eclipse Process Framework Project).

3.7 MAINTENANCE AND EVOLUTION

Two very important and related aspects of software engineering are software maintenance and evolution. After the software has been delivered (See Figure 8) and the software is in use the maintenance phase begins. Software maintenance is defined as:

"The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment." (Bennett & Rajlich, 2000)

Maintenance can be applied for a number of reasons, but in the literature one often finds the following types of maintenance activities (IEEE, 1990):

- Adaptive maintenance: changes in the software environment
- Perfective: new user requirements
- Corrective maintenance: fixing errors
- Preventive: prevent problems in the future

These types of maintenance are defined based on the nature of the change. An adaptive change is a change in the environment of the system. Therefore, the system must be adapted to fit in the changed environment. An example of such a change is an update of an underlying system. If a software product is depending on another system and this underlying system changes, the software must be adapted. Perfective changes are modifications on the software because the requirements of the users change. An example of such a change is a helpdesk system for an internet provider and the internet provider starts delivering new services. The software system should then be adapted to support the helpdesk with these new services. Corrective maintenance activities are related to errors in the system. Every software system contains errors and these should be fixed if they are detected and pose a problem for the users. Preventive maintenance activities focus on preventing future problems. An example of preventive maintenance is the addition of a stronger encryption algorithm in a banking system. The current encryption algorithm may be strong enough, but once an algorithm is broken it is too late to repair the system.

The term software evolution is used to indicate the phase of the software after it has been developed. Once the system is developed it starts evolving. Any successful software product will be used and maintained. During these cycles of maintenance the software evolves.

More detailed descriptions of maintenance activities and software evolution can be found in (Chapin et al., 2000). A more in depth characterization of maintenance activities and MDA is explained in (Seifert & Beneken, 2005)

3.8 CONCLUSION

In this chapter we described two parts of the software life cycle: the development phase and the maintenance phase. For the development phase many different models exist, we described two of them. We explained the waterfall model and the iterative model. Both processes we described: RUP and OpenUP are examples of an iterative process. Both RUP and OpenUP are not adapted to MDA but provide the means to extend the process to suit MDA. For instance, for MDA we described how a new role can be added that is responsible for meta-modeling and transformations. For OpenUP has been

an effort to implement an extension to support MDA. This extension includes new roles and activities. For both development processes tools are available that help customizing the process.

The second phase in the software life cycle is the maintenance phase. After a software system is delivered it enters the cycle of use and modification (See Figure 8). Four types of maintenance activities can be identified: adaptive, perfective, corrective and preventive. The overall process that starts after the system is delivered is called software evolution. All maintenance activities make evolve the system.

Chapter 4

4 CASE STUDY

This chapter describes the case study in which we rebuild an application using MDA. The first two sections of this chapter are devoted to the plans of the case study and the development environment we used. The other sections describe the architecture of the system and the architecture of the models and transformations..

4.1 DESCRIPTION

As a case for this study, we will use an existing business application developed at Getronics PinkRoccade. The application we rebuild is a simple business application that was originally designed to support the back office of the NEa (Dutch Emission authority). This organization controls the trade in CO_2 and NO_X emissions in the Netherlands. The system called Arend is used to administer CO_2 emission permits, complaints, objections, sanctions, external audits and NO_X trade. Besides the primary functions it also contains functions to administer incoming and outgoing mail, manage processes, manage users and maintenance of the archive.

The system is currently in use at the NEa and is maintained by Getronics PinkRoccade. Arend is a web application built on the Microsoft .NET platform and written in C# and ASP.NET. All documentation and source code of the system is available which can be used to speed up the development process. It was originally developed using the RUP and the developers did not apply any form of model driven development.

In the first stage of the case study, two use-cases from the original Arend are selected and implemented. In the second stage of the case study four changes are made to the implemented system to simulate maintenance activities.

4.2 DEVELOPMENT PLAN

The software development process that is used for this case study is the RUP. This process is used because we think it is currently the most widely used and accepted iterative software development process.

The case study contains two major components. The first component is the development of a system using MDA techniques. The second component is maintenance of the same system.

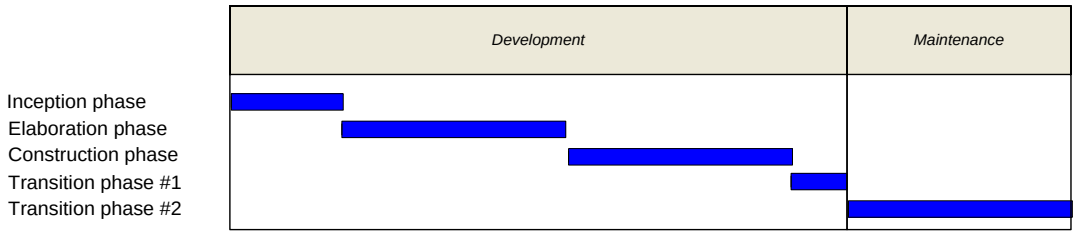


Figure 13: Project phases

Figure 13 provides an overview of the different phases and iterations of the case study. There are five iterations in total, divided among four phases. The inception, elaboration and construction phase all contain one iteration. The transition phase has two iterations. The first iteration will be used to finish the system and to deploy it. The second transition iteration is used to carry out some maintenance to the system.

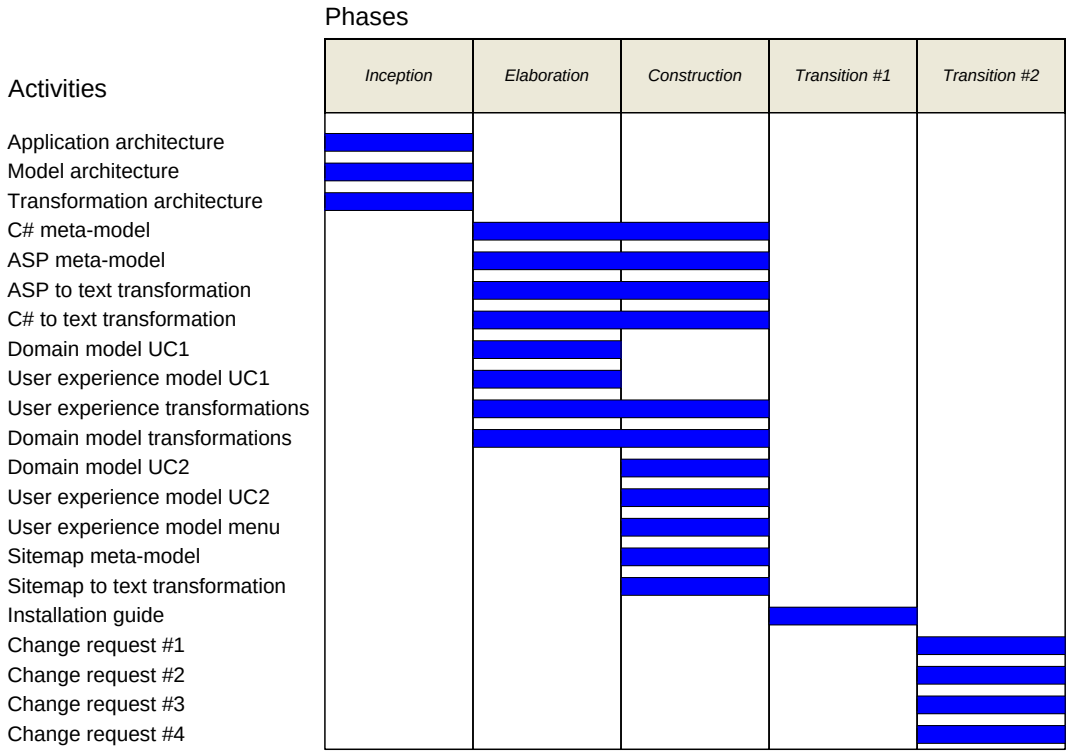


Figure 14: Project plan

The first phase of the project is the inception phase. In Figure 14 provides an overview of the larger activities per phase. In the inception phase the scope of the project is determined and the preparation for the next phases are made. For this project the inception phase is used to establish agreement on the scope of the project. In this project by selecting the use-cases to implement. The inception phase is also used to come up with an architecture and to setup the development environment. The architecture for the system is based on a reference architecture. The architecture for the models and transformations is derived from the system architecture. The reference architecture for the system is the existing implementation of Arend. Based on this three components are created: the application architecture, the model architecture and the transformation architecture. The application architecture describes the global structure of the application. The model and transformation architectures provide an overview of the models and transformations that are used to construct the application.

The purpose of the elaboration phase is to reduce risks. The largest risk in this stage of the project is related to the MDA approach. It is unclear whether it is possible to deliver a working system at the end of this phase. The creation of two UML profiles, two meta-models, two models and the transformations needed to generate the code seems too much work for one iteration. To ensure that the MDA approach is working for this project, a small use-case is selected and implemented in the elaboration phase. Only a minimal set of meta-models, models and transformations are built to implement the selected use-case. The set should be minimal but large enough to deliver a working system at the end of the phase.

Figure 14 shows the activities for the elaboration phase. In this phase the UML profile for the domain and user experience models and the C# and ASP meta-model are created. Based on the UML profiles a domain model and a user-experience model are created. These two models model the selected use-case. The transformations from UML to C# and ASP are created to implement the use-case. The transformation adds semantics to the models. The last step is to create the model to text transformation for the ASP and C# meta-models. This step implements code generation.

The third phase is the construction phase. In RUP projects, this is the phase where most of the work is done. Once the scope is established and the major risks mitigated, the bulk of the implementation is done in this phase. In this project the meta-models, models and transformations are extended to meet the requirements for the second use-case.

Figure 14 shows the activities for this phase. The ASP and C# meta-models are extended to support the second use-case. A new user-experience model is created to model the second use-case and the domain model is extended. The transformations are adapted to add semantics to the new models. The last step is to add a main menu to the system by implementing the sitemap meta-model and extending the user-experience UML profile and related transformations.

The fourth and last phase of the project is the transition phase. This phase contains two iterations. In the first transition iteration, the system is tested and deployed for use in production. In this case study we test the system and deploy it on a server to verify if it is not depending on the development environment. The second iteration of the transition phase is used to carry out the maintenance we planned. Four change requests will be implemented in this iteration.

Figure 14 shows the activities for this phase. In the first iteration a installation guide is created and tested. This is done to ensure the system can be deployed as a standalone system. In the second iterations four change request are handled to apply maintenance to the system. The nature of these changes remains unknown until this iteration starts.

4.3 DEVELOPMENT ENVIRONMENT

The application is built using a combination of Borland Together 2007 and Microsoft Visual Studio 2005. All models, meta-models and transformations are implemented using Together. Together has a solid foundation in the Eclipse framework. A large part of the tool is based on Eclipse and uses underlying EMF components to drive the modeling and transformation tools. Borland Together generates a plug-in that holds the meta-models and transformations for our project. This allows us to integrate our meta-models and transformations in the development environment.

Microsoft Visual Studio 2005 is used to build and test the generated source code. All generated code is imported in a project. Visual Studio was chosen due to the feature rich .NET platform. The .NET platform contains a number of ready-to-use components that reduce the amount of generated code.

4.4 APPLICATION ARCHITECTURE

The design of the system is based on a three tier architecture (see Figure 15). In this architecture three separate layers are responsible for separated tasks. The top layer is called the presentation layer. This layer is responsible for the user interface. This layer communicates with the layer in the middle, the business layer. The business layer contains the business and application logic. It checks the data against the business rules and contains the functionality of the application. The business layer communicates with the data layer. The data layer is responsible for storing and retrieving the data. This layer also allows to do some simple checks. The idea of this architecture is to separate concerns. Therefore, there is only communication between layers that are directly above or below each other. This architecture follows the layers pattern as described by (Buschman, Meunier, Rohnert, Sommerlad, & Stal, 1996).

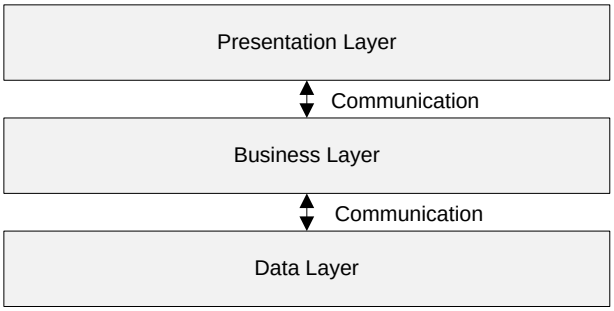


Figure 15: Three tier architecture of Arend

The application is built using the .NET platform. Therefore, ASP and C# are used for the implementation of the presentation layer. Each 'page' of the application is represented by a class. This class is split into two partial classes: a C# and an ASP class. The partial classes are written in different languages, but once compiled they form one class representing the page. The ASP code contains most of the layout information while the C# code contains most of the application logic. The presentation layer communicates with the business layer to display information on the screen. For a consistent look and feel, one master-page is used for the entire application. This master-page contains a menu that is used for navigation and a content panel. This content panel is filled with page specific content. The menu displays the contents of the sitemap. The sitemap is an XML format that contains a tree structure representing the application.

The business layer contains the classes that represent the objects from the domain of the system. In this case it contains entities like a document and a person. The business rules are also specified on this particular layer. In this application, the business layer is implemented in C#.

The data layer in this application is implemented by an object relational mapper. The object relational mapper is a library that handles all communication with the database. To add a class to the database it is enough to change their inheritance hierarchy and to add attributes to the fields of the class that must be stored.

Figure 16 shows a schematic representation of the structure of the application . This scheme shows how different components of the system are related with respect to implementation. The application

pages form the presentation layer. The business object are the business layer and DevExpress XPO is used to implement the data layer[6].

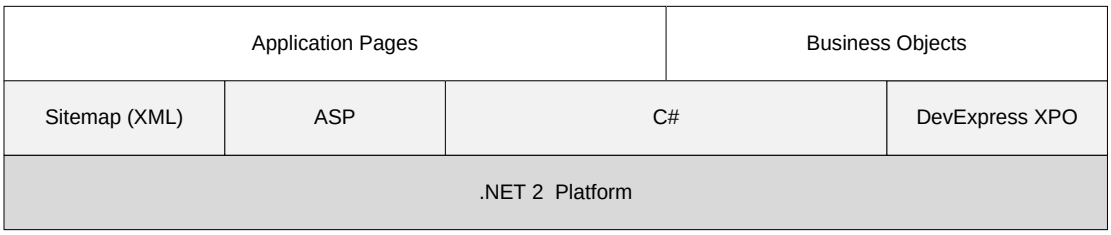


Figure 16: Schematic application structure

The schema (Figure 16) shows that the entire application is built on top of the .NET 2 platform. Sitemap is an XML format that is defined in the .NET 2 Platform. DevExpress XPO is the Object Relational Mapper (ORM) we used. ASP and C# are both languages from the .NET platform. The application pages are built using the Sitemap, ASP and C#. The business objects are implemented using C# and DevExpress XPO.

4.5 MODEL ARCHITECTURE

This section describes the model architecture of our system. With the model architecture we mean the overall organization and structure of the models and transformations that we use to build the system.

Based on the architecture of the system we can define a number of models that we need to build the system. At PIM level there are two models: the user experience model and the domain model. These models describe the three layers of the system. The user experience model describes the presentation layer. The domain model describes both the business and the data layer. Figure 17 provides an overview of all the models needed. The domain and user experience model are PIM models. At PSM level there are four models. The sitemap model, the ASP model for the user interface, the C# model for the user interface and the C# model for the domain. These models can be automatically derived from the user experience and domain model through QVT model transformations. There is a model transformation for each of these models. To transform the PSM models into code there are four model to text transformations. These transform the PSM into sitemap, asp and C# code. The transformations from PSM to code are implemented using a template engine called Xpand. This language provides the means to define a model to text mapping based on the meta-model. There are four transformation steps in Figure 17, but there are three Xpand transformations. There is a transformation from ASP to ASP code, from C# to C# code and from sitemap model to sitemap xml. The C# to code transformation is used twice. Once to transform the C# UX model to code and once to transform the C# domain model to code.

[6] It may seem strange that a library implemented in C# is actually schematically at the same level as C#. We think this is true in this case because the library exposes itself as a set of custom attributes. We interpret these attributes as a form of language extension.

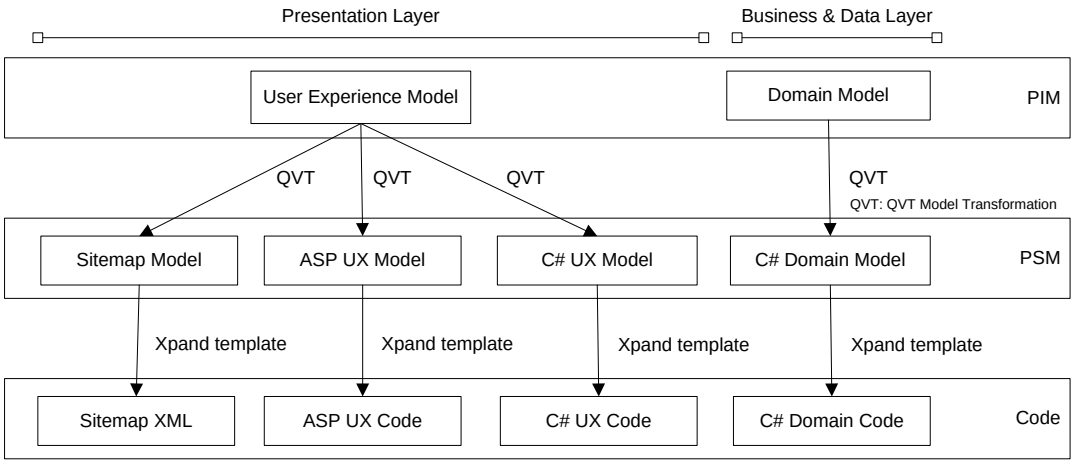


Figure 17: Schematic view of models and transformations

The two models at PIM level are modeled in UML 2.0, which is supported in Borland Together by default. To be able to model the PSM's we need meta-models for C#, ASP and Sitemap. Figure 18 provides an schematic overview of the model layer architecture. This picture shows the meta-meta-model, eCore, at level M3. At level M2 are the meta-models we use for this system. The UML 2.0 meta-model is included with Borland Together. The C# meta-model was partially included with the tool. The ASP and Sitemap meta-models were specifically build for this project. At Level M1, six models are shown with their instanceof relation to the meta-models. The user experience and domain model are instances of UML 2.0. The C# UX and Domain model are both C# models. The ASP UX model is based on the ASP meta-model and the Sitemap model is based on the sitemap meta-model.

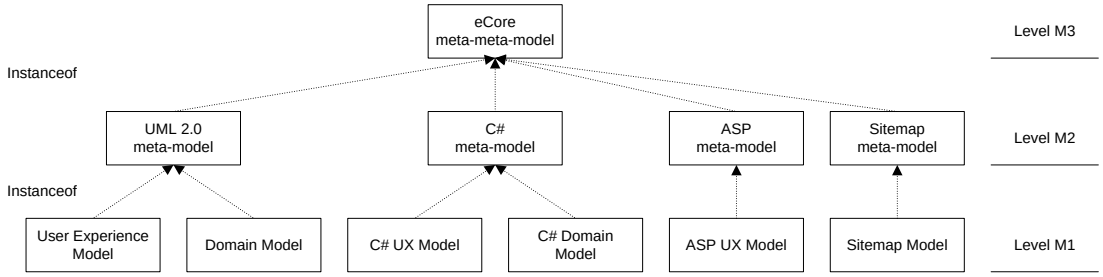


Figure 18: Modeling layers

The PIM's are represented using UML 2.0 and stereotypes. A User Experience model describes the pages in terms of structure, navigation and content. The domain model describes the business objects and their relations. In the domain model, stereotypes are used to determine which data is persistent. The PSM's are represented using custom meta-models, a C# meta-model for C# code, an ASP meta-model for ASP code and a Sitemap meta-model for Sitemap xml files. All meta-models, UML 2.0, C#,ASP and Sitemap are instances of the eCore meta-meta-model.

4.6 PLATFORM INDEPENDENT MODELS

The platform independent models are the most important models of the system. These models provide the input for the model transformations that will refine the PIMs into PSMs. The platform independent models describe the entire system on an abstract level. In this section we describe how and what we modeled at PIM level.

4.6.1 DOMAIN MODEL

One of the platform independent models for Arend MDA is the domain model. The domain model describes the business entities that play a role in the system. It describes the entities, their relations and how they are mapped to a database system.

The domain model is an UML 2.0 class diagram with an UML profile that can be used to handle persistency. The classes represent business objects with their properties. Associations can be used to define relations among different objects. Each of the classes modeled in the domain model will be transformed into a C# class representing the business object. The persistency profile determines what fields of the class are mapped to the database. The business layer is described by the domain model as well as the data layer. The business layer is directly modeled in terms of classes and the associations. The data layer is indirectly modeled trough the persistency UML profile.

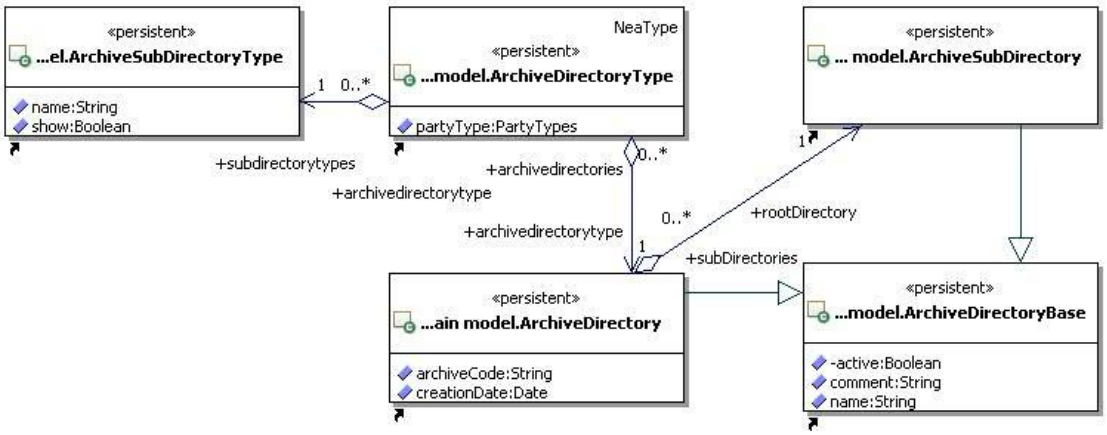


Figure 19: Domain model snippet

Figure 19 is a snippet of the full domain model. In this snippet the archive directory and subdirectory are modeled. These business entities are modeled as classes in the UML 2.0 class diagram. Based on this model the transformation can create a C# representation of this model. The associations in the domain model have both client and supplier role names. These names can be used in the transformation to create a variable name.

The UML profile that is used is created specifically for this project. This profile can be used to describe what classes or parts of classes must be mapped to a database. To do this the profile contains two stereotypes: “persistent” and “non persistent”. The first one indicates that a class must be mapped to the database; the second one indicates it should not be mapped. Because mapping complete classes may be too coarse grained, the stereotypes can also be used for attributes of classes. An attribute can have both stereotypes. If the class has the “persistent” stereotype, a single attribute can be removed from the database mapping by applying the “non persistent” stereotype for that attribute. If a class has the stereotype “non persistent” but one of the attributes has the stereotype “persistent” the class will be mapped to the database, but only the “persistent” attributes will be stored.

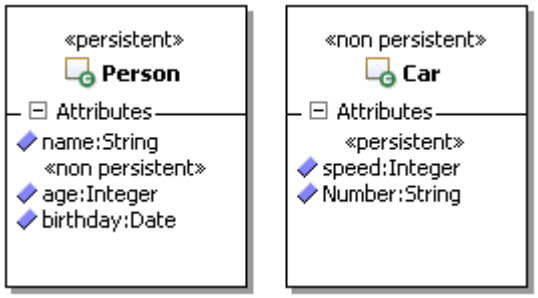


Figure 20: Examples of the persistency profile

Figure 20 provides an example of the persistency profile. The person class is marked with the "persistent" profile. Therefore the QVT transformation is able to create a C# class with attributes for the object relational mapper that indicate that the entire class except the age field must be stored in the database. The Car class demonstrates the "non persistent" stereotype. None of the fields of this class will be mapped to the database, except the speed field.

The full domain model can be found in Appendix C.

4.6.2 USER EXPERIENCE MODEL

The second model that forms the basis for Arend MDA is the user experience model. The user experience model is one of the two PIMs. A user experience diagram can be used to model the user-interface of a system. In this project, we used an user experience diagram to model the screens, the content of the screens, the navigation among screens and the menu structure. User experience models are modeled in UML 2.0 class diagrams with an user experience profile. This profile contains three stereotypes to model screens and the content of screens. The following stereotypes can be used: "screen" , "input form" and "content bundle". This profile is based on the user experience profile as described by (Kozaczynski & Thario, 2002).

The main components of a user experience diagram are screens. A screen is modeled as a UML class with the stereotype "screen". Screens represent the user interface of the application. A screen has a title, content and operations. The name of the class represents the title of the page. The content can be modeled by adding fields to the class or by associating the screen with a "input form" or "content bundle" class.

A screen can have two types of content: input forms or content bundles. Input forms are meant for data entry, content bundles for display. An input form is modeled as a UML class with the "input form" stereotype. The attributes of the class represent inputs on the form. The type of the attribute determines what kind of input is modeled. For example, an attribute "Name:String" is the model for a text input. Figure 21 is an example user experience diagram in which a screen is associated with an input form. Figure 22 represents the screen that is modeled by Figure 21. The class "Add archivetype" models a screen with the title "Add archivetype". The operations of the screen class become buttons on the screen and the associated input form is displayed on the screen. The "Archive" class models a form with the title "Archive". Each of the attributes models an input on the form.

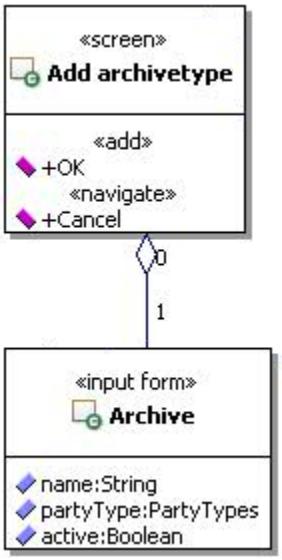


Figure 21: User experience diagram of a form

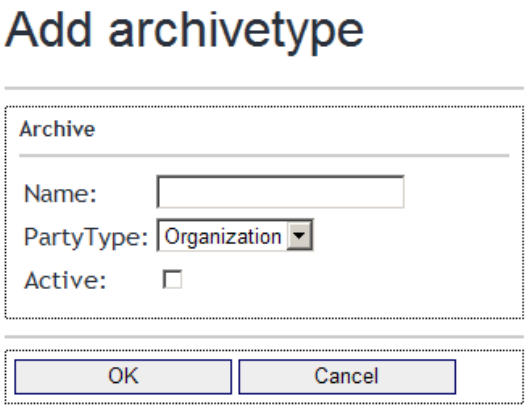


Figure 22:Single form example screen

The second type of content that can be added to a screen is a content bundle. A content bundle is a class with the "content bundle" stereotype associated with the screen class. Classes with the stereotype "content bundle" model screen content that displays information. A content bundle is a region on the screen with a title and text that displays information. The attributes of a content bundle class represent the labels. For instance, an attribute called "name" will represent a label with the text "Name: <value>". Figure 24 provides an example of a screen with a content bundle.

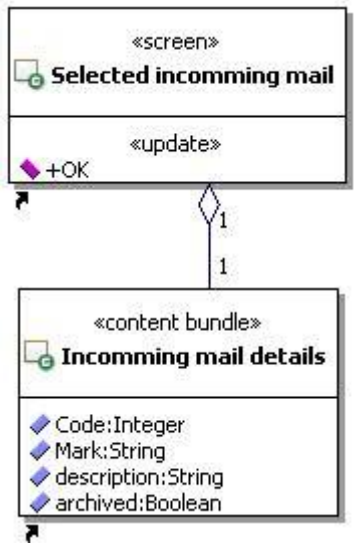


Figure 23: User experience diagram of a content bundle

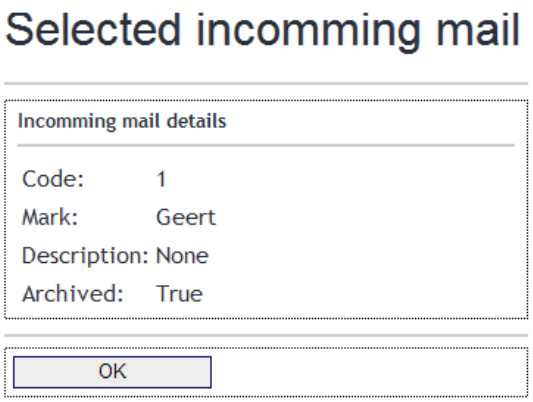


Figure 24: Example of a content bundle

Modeling the content of a screen is done by associating content with a screen class. The multiplicity of the association between a screen and an input-form or content-bundle models the type of content. A one-to-one association models a single instance of the content. Figure 23 is an example of a one-to-one association. If the association is one-to-many, it models a table of content. If a content bundle is associated to a screen with a one-to-many association it models a table. The attributes of the content bundle class represent columns of the table. If an input form is associated to a screen with a one-to-many association it models a table with input capabilities. The attributes of the input-form class represent the columns of this tables and the operations model operations on the table.

An example model of a table with input capabilities can be found in Figure 25. It models a page like the page in Figure 26. The columns “Name” and “Show” are modeled by the attributes of the input form. The operations of the input form result in two extra columns. An “Edit” column that can be used to toggle an editor. The “Delete” column can be used te remove a record from the table. The “Add” operation results in a small form below the table. This small form can be used to add a record to the table.

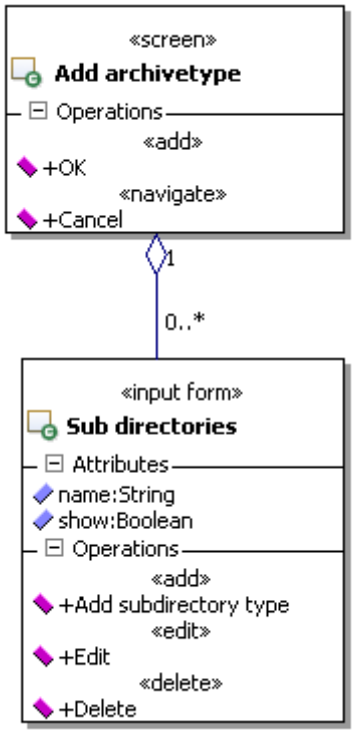


Figure 25: User experience diagram of a multiple input form

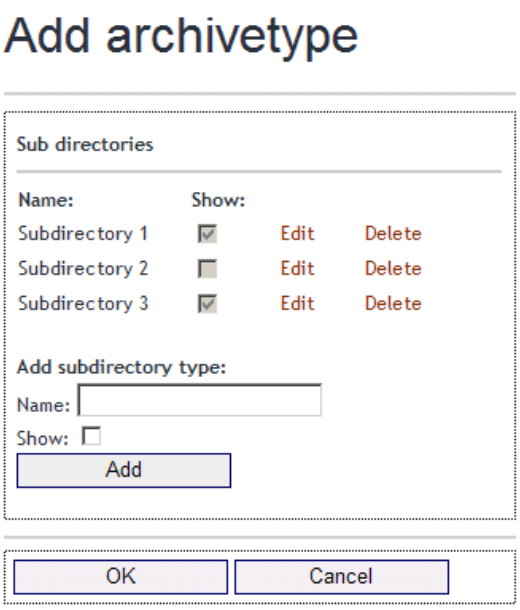


Figure 26: Multiple form example screen

Operations on screen classes have stereotypes that determine what type of action must be executed when the button is activated. The simplest stereotype is no stereotype. This means the operation represents a navigation action. A directed association with the name of the operations indicates to what screen this action navigates. Other stereotypes include “add”, “edit” ,”update” and “delete” actions. The semantics of these actions are defined in the model transformation.

If an operation is a navigation operation it will search for a directed association with another screen that carries the name of the operation as button. This directed association indicates to what screen the system must navigate. Figure 27 provides an example of navigation operations. The first screen “start” contains an operation “Start”. There is a directed association between the first screen and the screen in the middle “A screen”. Pressing the “Start” button will navigate to the screen in the middle.

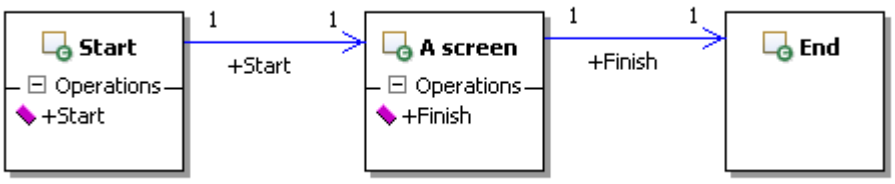


Figure 27: Navigation example

The user experience model can also be used to model the menu structure of a system. The stereotype “menu” and “submenu” can be used to model a menu structure. A menu is a UML class with the stereotype “menu”. A menu can have submenus. These are modeled with UML classes that have the “submenu” stereotype. Directed associations are used to add a submenu to a menu. The submenus can contain operations. Each operations corresponds with a directed association between the menu structure and one of the screens. Operations in a menu class are always navigational operations (at least for this project). Figure 28 is a snippet of our menu model. It defines one menu with one submenu. This submenu contains operations that navigate to a screen.

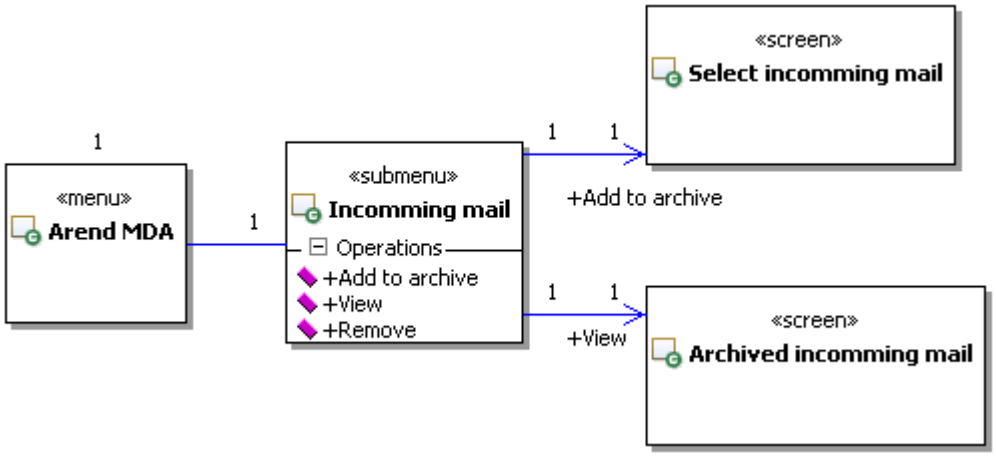


Figure 28: Snippet of the menu structure

Figure 29 is a screenshot of the menu that is modeled with the structure from Figure 28. The “menu” and “submenu” classes are mapped to a tree structure that contains the operations. In this project we choose to use a tree. Other menu structures can also be modeled.

Figure 29: The modeled menu structure

The full user experience models can be found in Appendix D and Appendix E.

4.7 PLATFORM SPECIFIC MODELS

To develop the Arend system we used three meta-models to model the platform specific models. We used the C#, ASP and Sitemap meta-model. In this section we introduce each of the meta-models and discuss their usage and their structure.

4.7.1 C# META-MODEL

To model the PSM level of our system we need a meta-model of our platform. Our platform consist of four different domains. The platform contains ASP, C#, Sitemap and the attributes for the ORM tool. The C# meta-model is used to model the C# code of our system.

For Arend MDA we adopted a partially built C# meta-model. This meta-model is provided by Borland as an example meta-model for Borland Together. We extended this meta-model to a more detailed meta-model. The meta-model supports most of the structural aspects of C#. It is able to model namespaces which can contain other namespaces and classifiers. Classifiers can be enumerations, classes, structures, interfaces and delegates. The meta-model does support structural aspects of C# like methods, fields, attributes and properties with getters and setters but it does not support statements and expressions. To be able to specify the implementation of methods, getters and setters a special meta-model element is introduced. This element can hold plain text to allow generation of statements and expressions in the C# meta-model.

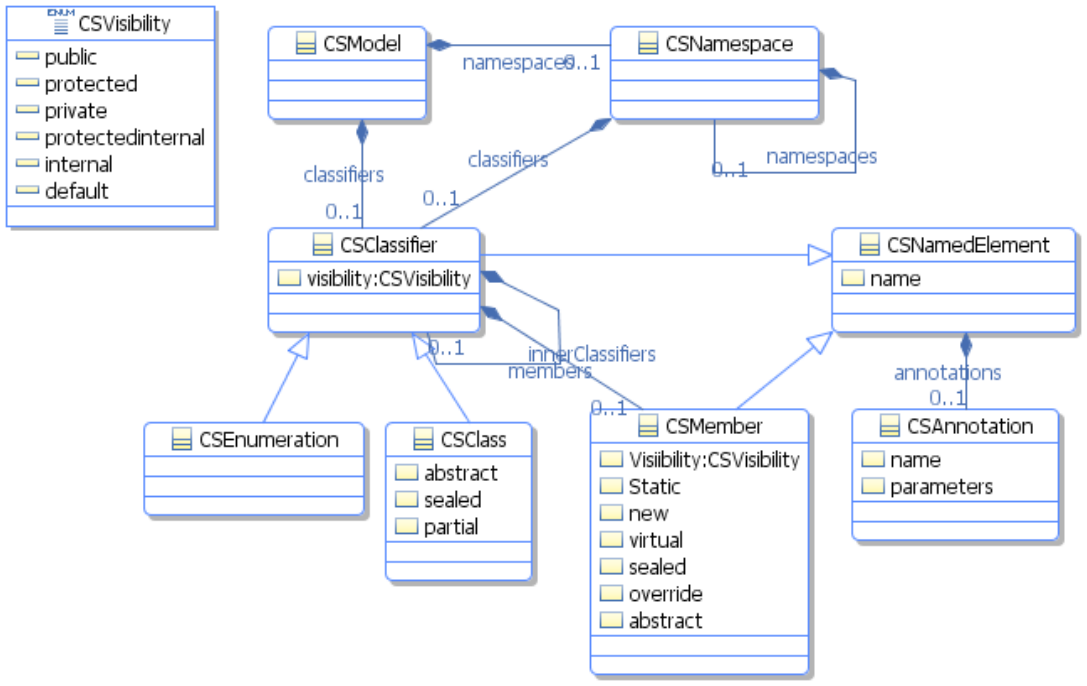


Figure 30: A part of the C# meta-model

Figure 30 is a diagram of a part of the C# meta-model. This diagram provides a coarse grained picture of the meta-model. It shows the model with namespaces and classifiers. A namespace itself can contain namespaces and classifiers. A classifier can be an enumeration or class. In the full meta-model other classifiers are modeled. A classifier can have members. In this diagram only the abstract type member is shown. A possible specialization of a member is a method. All classifiers and members are generalized as named element. This named element contains annotations. These annotations are called attributes in C#, but this term is often confused with C# fields. Therefore, we stick to the name annotation. A C# annotation is a declarative piece of text that can be added to all named elements.

4.7.2 ASP META-MODEL

As described in the previous section, the platform for our system contains multiple domains. One of these is ASP. The ASP meta-model is used to model the ASP code of our system.

To support modeling ASP.NET code in Borland Together an ASP meta-model is created. The basis for the ASP meta-model is a simple structure of two abstract types: pages and tags. Both types have multiple specializations. There are two page types: a normal page and a master page. A master page is a concept from the .NET platform and represents a template page for normal pages. A normal page fits inside a master page and provides the content for the template.

Both page types have a body that can contain tags. Tags represent ASP and HTML tags that can be used to model web pages. Many html tags are supported to provide layout and style. The ASP tags represent controls that can be placed on pages and reflect the ASP controls available in .NET.

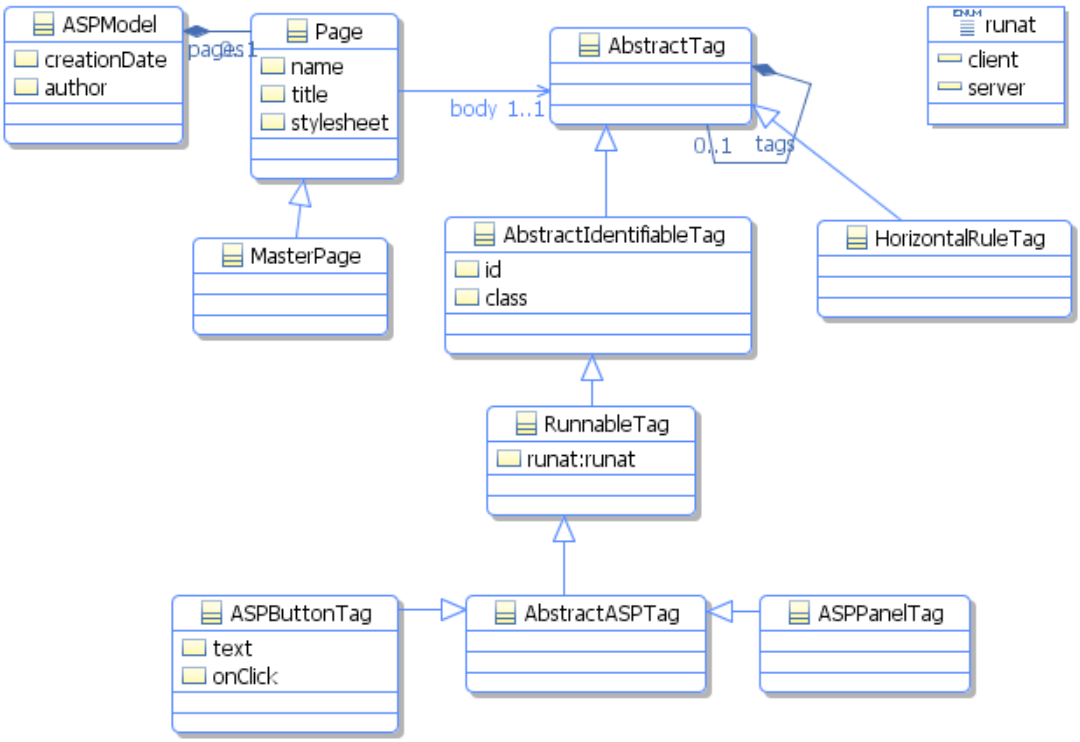


Figure 31: A part of the ASP meta-model

Figure 31 provides a part of the ASP meta-model in UML notation. This picture shows the design of the ASP meta-model. The root model element is the ASPModel. This element contains pages. Each page represents an ASP page. Pages have a body. A body is always a tag. Tags are modeled trough a system of specialization. The most abstract type is the abstract tag. Simple HTML tags are modeled directly as a specialization of the abstract tag. An example is the horizontal rule tag: “<HR/>”.

4.7.3 SITEMAP META-MODEL

The simplest model used in the development of Arend MDA is the sitemap meta-model. The meta-model is based on the structure that the .NET platform uses to specify sitemaps. Sitemaps in .NET are XML files with a sitemap schema. In the simplest form a sitemap contains just two types of nodes. Both are modeled in the sitemap meta-model.

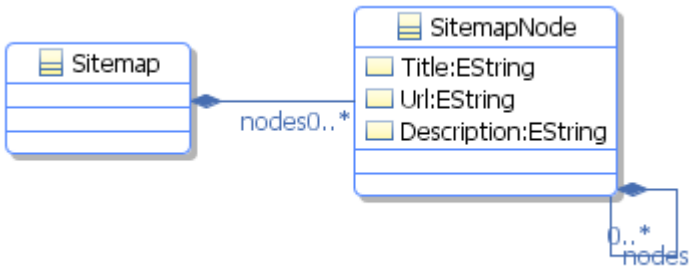


Figure 32: Sitemap meta-model

Figure 32 provides the full Sitemap meta-model in UML notation. This model is a very simple model. The root element is a Sitemap element. This element can contain multiple SitemapNode elements. Each SitemapNode element has a collection of nested SitemapNode elements. This can be used to create a tree structure of nodes.

4.8 PIM TO PSM TRANSFORMATIONS

This section describes the model transformations from PIM to PSM. The first subsections contains an overview of the structure of the transformations and meta-models. In the section that follow we introduce the transformations from user experience to ASP and C#, and we introduce the transformations from domain model to C#. The last section describes the language we used to implement the model transformations.

4.8.1 TRANSFORMATION ARCHITECTURE

The transformation architecture gives an overview of the transformations needed to refine the abstract models into more concrete models of the application code. Figure 33 provides an overview of all models and transformations. The white boxes represent models and the white ovals represent transformations. These are directed transformations from the source to the target model. The two source models, the user experience and domain model, are transformed into four different models. The user experience model is transformed into the ASP code, the C# code behind files and a sitemap for the main menu. The Domain model is also transformed into a C# model, but using a different transformation than the user experience model. This is due to the difference in semantics of both models. All m2m transformations are implemented using QVT operational mappings. The m2m transformations are documented using the transformation pattern in 0.

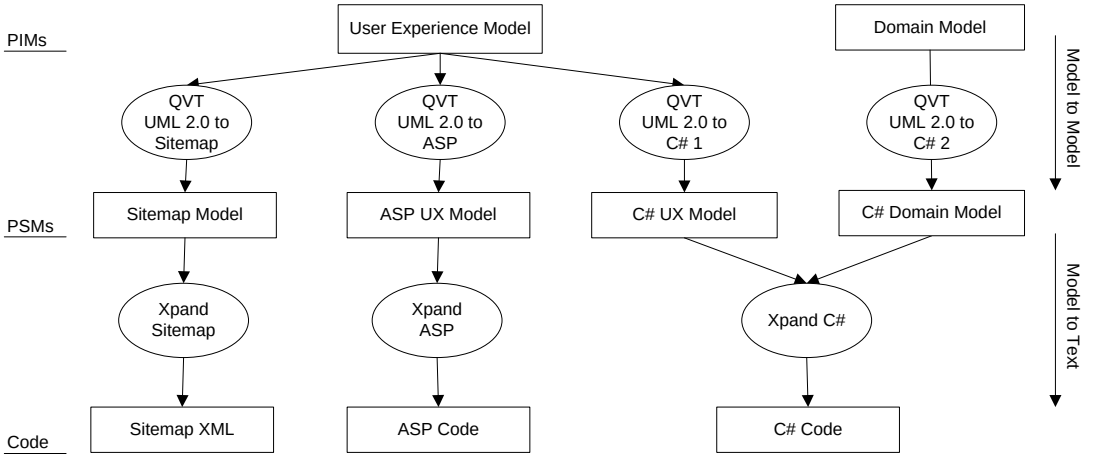


Figure 33: Transformation overview

The target models of the first series of transformations, the PIM to PSM transformations, become the source models for the second series of transformations. In the second transformation step, the PSM to Code transformation, these models are transformed into a textual representation. This is implemented using Xpand templates.

4.8.2 USER EXPERIENCE MODEL TRANSFORMATIONS

In total there are seven transformation steps needed to transform the models into working code. The four most important transformations are the transformations from user-experience and domain models to ASP and C# models. The transformations contain the semantics for the models. All model transformations can be described using the transformation pattern (see section 2.4).

The transformation of the user-experience models is split up in two parts. One part is responsible for transforming the model into an ASP model describing the structure and layout of the user interface. The transformation follows the pattern as illustrated in Figure 34. The user experience model is an instance of the UML 2.0 meta-model and is transformed by a QVT transformation. The target model is an instance of the ASP meta-model. The transformation creates a set of pages, a page for every class with stereotype “screen” in the source model.

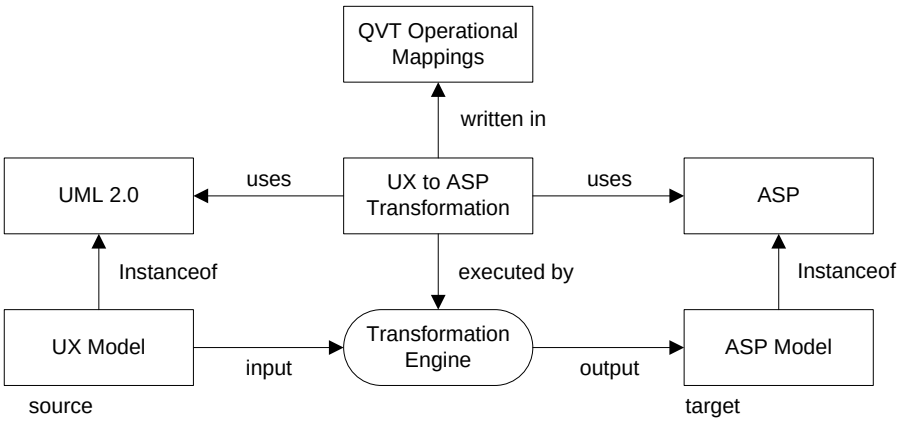


Figure 34: Pattern for the user experience to asp transformation

The second part of the user-experience model transformation is responsible for the C# code that runs server side and that implements the application logic behind the ASP pages. The C# files contain the

code that loads the data and fills the input forms, handles events from the user interface and implements the actions. The user-experience to C#-model transformation creates a C# class for each class with the stereotype “screen” in the user-experience model. The transformation adds event handlers for each operation and the code that implements the operations. The event handler that is hooked to the load page event handles data retrieval. The pattern for this transformation is illustrated in Figure 35.

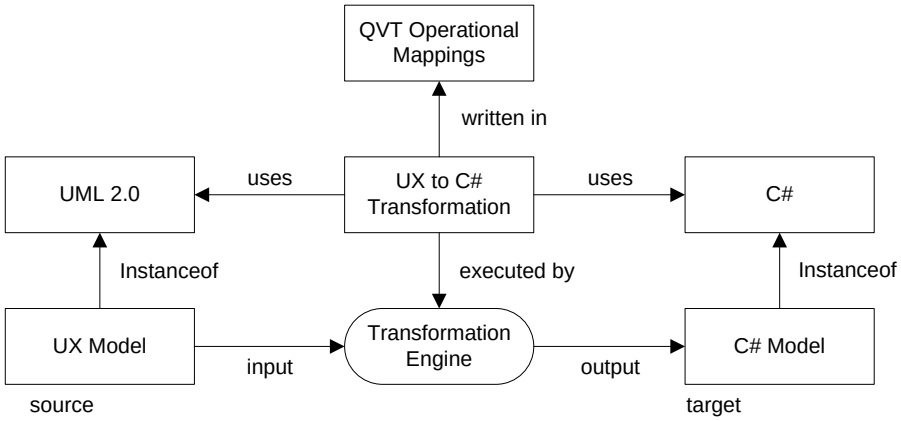


Figure 35: Pattern for the user experience to C# transformation

The last and smallest step in the transformation process is the creation of the sitemap. The sitemap is a model that is used to model the sitemap file. The sitemap file is used to drive the main menu of the application. The sitemap model is created by a transformation that transforms all classes with the “menu” and “submenu” stereotypes into a sitemap model. This transformation is illustrated in Figure 36.

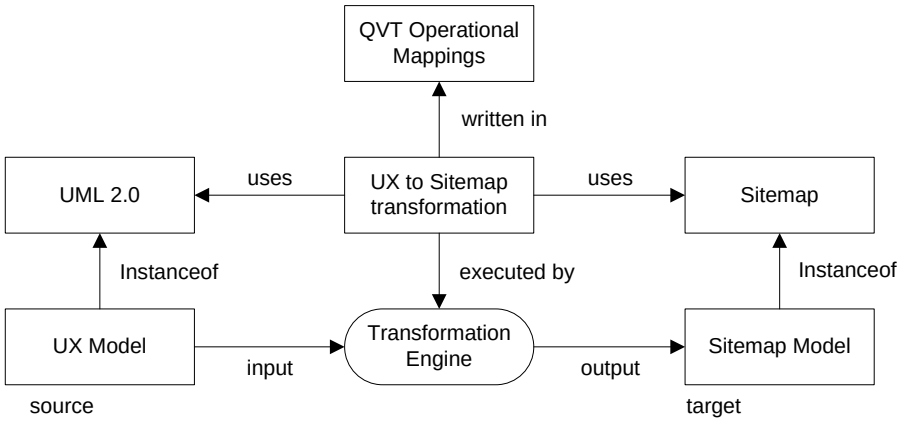


Figure 36: Pattern for the user experience to Sitemap transformation

4.8.3 DOMAIN MODEL TRANSFORMATIONS

The transformation from domain model to C# model is the transformation that is responsible for creating the business layer (see Figure 15). The business layer contains classes that represent the domain model and handle storage and retrieval of the data. The business layer should also contain the business rules. In this case business rules are not included in the transformation. Business rules can be specified using OCL constraints in the UML 2.0 models, but transforming the OCL constraints to C# code is beyond the scope of this project. OCL support for C# is available (Arnold, 2004) but need to be adapted to work with our transformations. The transformation from UML 2.0 to C# is also

responsible for naming all the variables. If the transformation renames a variable from A to A' the OCL expressions should also be parsed and adapted to these changes. Therefore implementation is possible but not within the scope of this project.

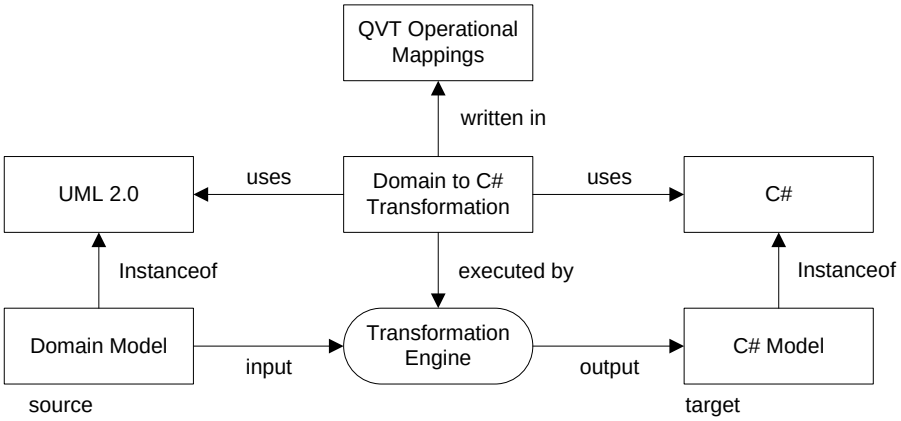


Figure 37: Pattern for the domain model to C# transformation

The Domain to C# transformation contains two parts. The first part is the structural transformation which is a one-to-one mapping of UML 2.0 classes to C# classes. For every package a namespace is created and the content of the package is transformed into content for the namespace. Each UML 2.0 class is transformed into a C# class, and UML 2.0 enumerations are transformed into C# enumerations. The pattern for this transformation is illustrated in Figure 37.

The interesting part is the part of the transformation that handles the persistency stereotypes. When the transformation transforms UML properties into C# properties, it checks the stereotypes. Depending on the type of the property and persistency stereotypes of the property and the containing class, it transforms the property into a C# property with the needed attributes. If a containing class is marked "persistent" or the property itself carries this stereotype the transformation will add a "persistent" attribute to the C# property. An example of such an attribute can be found in Code section 3. In this code section a persistent property archiveCode is illustrated. The persistent attribute can be found on line 3.

```
1  private string _archiveCode;
2
3  [Persistent("archiveCode")] //This is the attribute
4  public string archiveCode{
5       get {
6             return _archiveCode;
7       }
8
9       set {
10              archiveCode = value;
11      }
12 }
```

Code section 3: Example C# property

The transformation of associations is the most complex part of the transformation. Two important properties of the source model determine what will be created in the target model. For each UML 2.0 class in the source model the transformation looks up all incoming and outgoing transformations[7].

[7] Not all incoming associations can be found due to a bug in the QVT interpreter shipped with Borland together 2007. The 'allinstances' statement should return all instances of the specified type, in Borland Together the output depends on the context of the call. Therefore, the implementation of the transformation might not include all incoming associations. Outgoing associations are accessible from the owner and therefore not a problem in the transformation.

Depending on the multiplicity a single value or a collection property is created. The type of the association is encoded in an attribute for the property. If the association has the aggregate type an attribute “Aggregated” for the property is added. This is illustrated in Code section 4 at line 1.

```
1  [Association("AssociationsubDirectories",typeof(ArchiveSubDirectory)),Aggregated]
2  public XPCollection<ArchiveSubDirectory> subDirectories{
3  get {
4              return GetCollection<ArchiveSubDirectory>("subDirectories");
5       }
6  }
```

Code section 4: Example C# collection property

4.8.4 QVT TRANSFORMATIONS

The domain and user experience models are transformed to C# and ASP models by three QVT transformations. All three transformations are written in QVT Operational Mappings. There is one transformation that transforms the domain model to a C# model and two transformations that handle the user experience model. The user experience model is transformed to two separate pieces, the ASP part that contains the layout information and a C# part that contains the logic for the user interface.

Besides the three transformations there are a number of QVT libraries. These libraries contain a number of queries that are used throughout all three transformations. One of the libraries contains the naming conventions. The naming conventions are queries that provide names for all kind of types in the models. For instance, a textbox or a dropdown identifier must be mapped based on a UML property. Another library contains extra queries that extend the UML meta-models with extra functions. One library defines persistency. When this library is included in QVT it adds a method isPersistent() to the UML class model element. This makes development of the transformations easier and simplifies changes. Code section 5 contains an example piece of QVT code to illustrate the language.

```
1  mapping uml20::kernel::Enumeration::toCsharpEnumeration() : csharp2::CSEnumeration
2  {
3       Name:=self.name;
4       Visibility:=self.visibility.oclAsType(uml::kernel::VisibilityKind).toCsharp();
5       Members:=self.ownedLiterals->collect(liter | liter.toCsharp())->asOrderedSet();
6  }
```

Code section 5: Illustration of the QVT Operational Mappings language

4.9 CODE GENERATION

Both the ASP model and the C# models can be transformed to text using a model to text transformation. For Arend MDA we implemented two Xpand templates that convert the C# and ASP models to text files. The templates are wrapped in a Borland Together plug-in which enables us to call the template for each ASP page and C# class. Every template call results in a generated text file which is saved to the file system. The filename is determined using an Xtend expression[8]. Xpand is a very simple language to define model to text transformations. It contains a very limited but sufficient number of constructions that allow the programmer to write transformations rules for each type defined in the source meta-model.

[8] Xtend is the expression language that can be used inside Xpand templates.

The following example code shows a transformation rule named “buildTag” for model elements of type “ASPLabelTag”. The definition contains plain text, which will be outputted by the transformation engine and an expand call. The expand recursively calls the “buildTag” rule for all contained tags. All text between ‘«’ and ‘»’ are Xpand constructs. Code section 6 provides a sample of Xpand to illustrate the Xpand language.

```
1  «DEFINE buildTag FOR ASPLabelTag»
2  <asp:label«EXPAND buildAspArguments FOR this»>
3       «EXPAND buildTag FOREACH this.tags»
4  </asp:label>
5  «ENDDEFINE»
```

Code section 6: Illustration of the Xpand template language

4.10 CONCLUSION

In this chapter we described the plan for our case study and how we executed the plan. The case study contained two different overall parts: a part to construct the system and a part to apply some maintenance to the same system. For this case study we used RUP to structure our project and MDA to build the system. The combination of these two resulted in a project with four phases. An inception phase in which we determined the scope, architecture for both our system and the tools to build our system. In the elaboration phase the MDA approach was used to implement the first use-case. To implement this use-case we had to build two meta-models: ASP and C#. We also created the UML profiles for the domain model and the user experience model. We defined the transformations between the phases and created the model to text transformations. In the third phase, the construction phase we implemented another use-case. To implement this we had to extend our existing meta-models and implemented another meta-model: Sitemap. The transition phase was divided into two iterations. The first iteration was used to deploy the existing system and to see how it performed. In the second iteration of the transition phase we applied some maintenance. Four change request were created by a third party. These were implemented to see how maintenance can be applied in a project that has been developed using model driven techniques.

Chapter 5

5 EVALUATION

The evaluation of the case study contains two parts. The first part is the evaluation of the software development process. In this part each of the phases of the software development process are discussed. We start with the inception phase and evaluate each of the goals for this phase. Besides the goals we also describe our observations. The issues that are observed are described in the next part of the evaluation. In the second part of the evaluation we enumerate all the issues. For each issue we provide a description of the problem and if necessary examples to clarify the problem. We also provided a possible solution if the solution was known to us.

5.1 SOFTWARE DEVELOPMENT PROCESS

The evaluation of the software development process is based on the phases of the RUP. For each phase a subsection describes the goals and the results. For all phases the goals are evaluated. The goals come from the iteration plans we created during the project.

Figure 38 provides an overview of the evaluation. Each of the phases of the software development process is defined in a separate section. The transition phase is split in two sections. The first section describes the evaluation of the deployment of the system. The second iteration describes the evaluation of the maintenance activities.

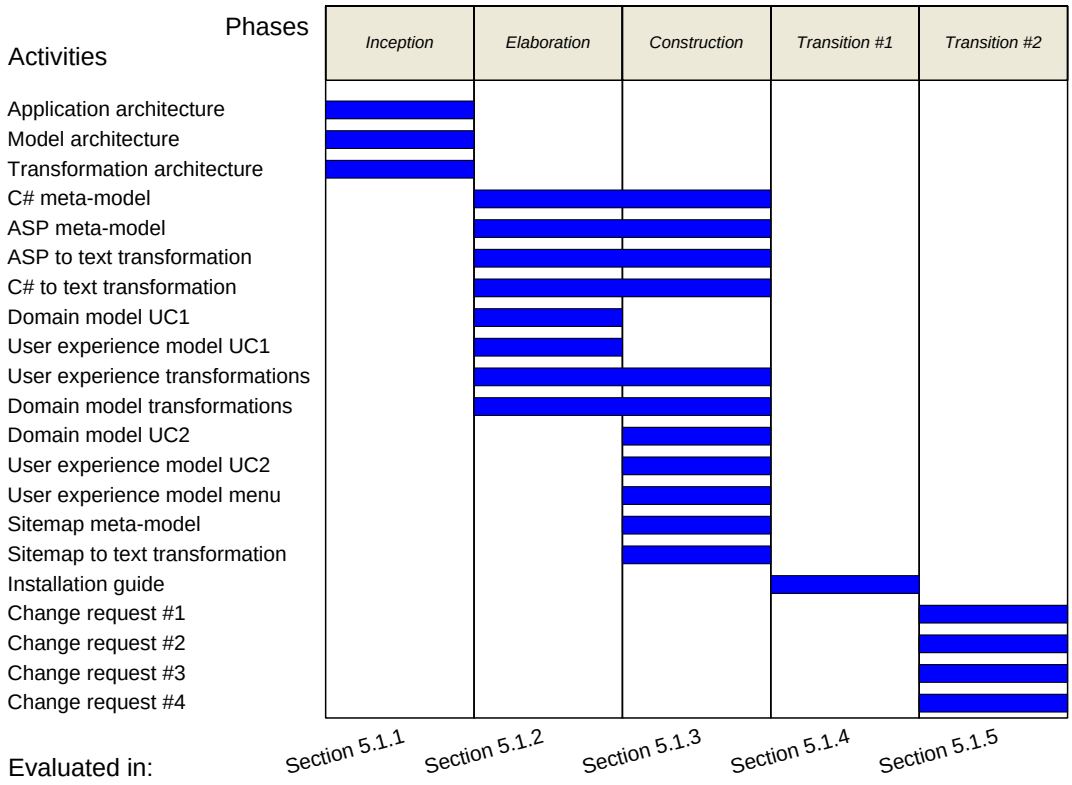


Figure 38: Evaluation map

5.1.1 INCEPTION PHASE

The inception phase of our project is used to setup the initial project and to find out how the project should be carried out. For this phase we had four goals. The most important goal was determining the scope of the project. The other goals are related the architecture of the system, the approach to use and the development environment.

Goal: Determine scope of the project

The scope for this project was based on the original Arend project. Due to the amount of time available a small selection of use-cases was made. Two use-cases were selected, one simple use-case for the elaboration phase and a more complex use-case for the construction phase. The following two use-cases were selected:

UC1: Archive maintenance. Adding, editing and removing archives.

UC2: Archive mail items. Putting incoming and outgoing mail items in one of the archives.

Goal: Determine architecture

Based on the selected use-cases it was now important to come up with an architecture and an approach to implement the use-cases using MDA. Because the existing system was available it was possible use this as a reference architecture. The MDA approach was defined using the reference architecture as guide.

Goal: Determine approach

To build the system using MDA we needed to find out what models we needed and how the transformations would transform the models. In the inception phase we determined that we needed the user experience and domain model at PIM level and the ASP and C# models at PSM level.

Goal: Setup development environment

We wanted to use Borland Together 2007 and Visual Studio 2005 as our development environment.

Observations

In the early stages of a project it is important to select the proper tools and think about the technologies one may need during development. Tools may claim they support MDA or at least that they implement some of the technologies, the question is what parts of the standard are implemented. In the case of Borland Together the major part of the QVT specification is implemented and some tools are provided to create custom meta-models. The problem is that not all parts of the tool are well documented and sometimes contains bugs.

Another observation we made during this phase is that in fact two architectures need to be build. One for the system under development and an architecture for the tools that develop the system. The architecture for the tools to develop is needed because MDA only provides the techniques to apply model driven engineering. MDA does not specify what meta-models are needed and what model transformations are going to be used to build the system. One of the goals for the inception phase is the setup of the working environment. Part of this setup includes documenting the MDA approach: what meta-models, models and transformations are needed to build the system.

5.1.2 ELABORATION PHASE

According to RUP the elaboration phase is the phase to mitigate the risks and to find out if the project is feasible. In this case study we used the elaboration phase to make sure that our approach was working and that we were able to create a working system. In this section we evaluate the goals for the elaboration phase.

Goal: Stabilize scope and architecture

The architecture created in the inception phase was tested in practice in the elaboration phase. The implementation of the first use-case (UC1) was a test for both the architecture of the system and for the MDA approach. The architecture for the system was based on the reference architecture provided by the original Arend project. There were no problems building the system according to the architecture. We did have some minor problems with the ORM tool. Creating persistent classes in C# was more work than expected but worked fine in the end.

Goal: Implement models and transformations for first use-case

The major part of the work in the elaboration phase was creating the meta-models and transformations. Borland Together provided a basic C# meta-model and an Xpand template as demonstration of the DSL toolkit. We used the Borland DSL toolkit to create a plug-in that contained our ASP and C# meta-model and the Xpand templates to transform the models into text. The meta-models were designed as minimal implementations of their domain. With a minimal implementation of a meta-model we mean that the scope of the meta-models was our project and not the complete domain. The meta-models contained the model elements we needed to model our system. The ASP

meta-model only contained the tags that we needed for the implementation of the use-case, and the C# meta-model only contained the structural elements of C#. Expressions and statements were not supported.

The next thing we did was building the UML models. Based on the use-case descriptions two models were created. Both models are UML models contained class diagrams, a diagram for the domain model and a diagram for the user-experience model.

The transformations from UML 2.0 models to the ASP and C# models are implemented in QVT Operational Mappings. Creating these transformations showed all flaws in both the source as the target meta-models. It was clear that not all features for the C# and ASP meta-model were available. This meant changing the meta-model, rebuilding the plug-in and reinstalling the plug-in before the new features became available in Together. Creating the meta-models was a time consuming process. The main cause for this was the poor quality of Borland Together. Many bugs frustrated our work.

The model transformations created ASP and C# code for the domain model and the user experience model. The transformations were based on small proof of concept implementations of the model elements. Once the transformations finished, the generated models were transformed to code using the Xpand templates. The generated code was imported in Visual Studio 2005 and compiled.

Goal: Deliver working system

Compiling the system using Visual Studio 2005 was very successful. The first time a Visual Studio project was created and the files were imported by hand. The output directory for the Xpand engine was set to the project directory. This way Visual Studio imported the generated files automatically once the files changed. The system was compiled and debugged in Visual Studio.

Observations

We developed our own meta-models as we needed additional features. We started with a very basic meta-model for ASP and while developing the transformations to ASP we added model elements. Adding model elements was not a problem, removing model elements or changing elements resulted in problems. We will refer to this problem as "***Model and Meta-model co-evolution is difficult***".

One of the problems we encountered during this phase is that the complexity of model transformation increases as the structural difference between source and target meta-models increases. We will refer to this problem as "***Structural incongruence increases transformation complexity***".

During the implementation of the QVT transformations it became clear the some of the object oriented techniques we used in the meta-models were not supported by the QVT language. For instance, a mapping that has a abstract return type in the signature is not allowed. Even if the implementation of the mapping does return concrete types. We will refer to this problem as "***Lack of object orientation in QVT***".

A detailed discussion of these issues can be found in section 5.3.

5.1.3 CONSTRUCTION PHASE

In the construction phase a larger and more complex use-case was selected for implementation. In this section we evaluate the goals we set for this phase of the project.

Goal: Analyze, design and implement functionality for the complete system

To implement the second use-case we extended the domain model and added a user-experience diagram to the user experience model. That made it possible to transform the use-cases independently. With the implementation of the second use-case the number of screens in the system increased. To cope with the larger number of screens we added a menu to the system. A new diagram was added to the user experience diagram to model this main menu.

The basis for the menu is formed by a separate user-experience diagram. This diagram contains classes with the menu stereotype and associations between menu classes and screens. A QVT transformation transforms this model into a sitemap model. The sitemap model is a very simple eCore model that can be transformed to XML by an Xpand template. This XML file is read by a special sitemap component on the web pages.

Implementing the sitemap component on every page seemed like a simple solution. Some changes in the user experience to ASP transformation resulted in a sitemap on each page. However, the .NET platform also offered a solution that enabled us to create a master page for each page containing the menu. The master page contains the menu and a content holder that can be filled with page specific content. Because we think it is a good thing to take full advantage of the frameworks we use we choose to implement the master page. This made the generated code smaller, but the meta-models larger. The ASP meta-model was adapted to support master pages and content holders.

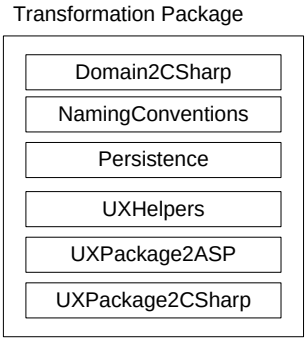


Figure 39: Transformation package elaboration phase

Due to the implementation of the second use-case the number of transformations increased. New transformations were added for the sitemap, the menu and the creation of the master page. The complexity of the transformations themselves also increased due to new features required to implement the second use-case. This resulted in the need for a better structured set of transformations. The package of transformations from the elaboration phase is shown in Figure 39. We decided to put our transformations in a structured set of QVT packages. Packages for the user experience, domain model and sitemap transformations were created to separate concerns. Global mappings, like mappings to handle naming conventions, were put in a separate package called common. The new structure of the packages is shown in Figure 40.

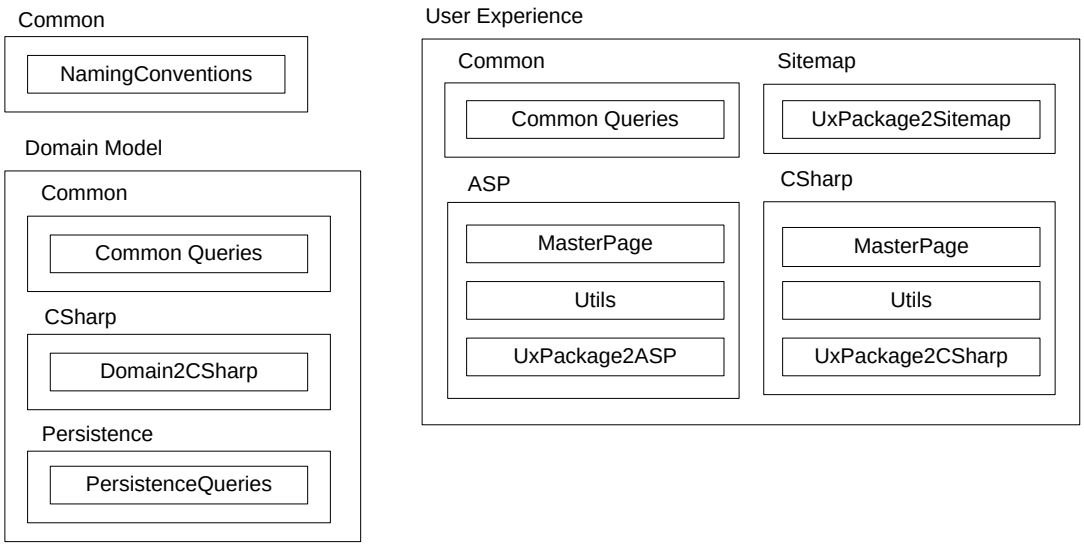


Figure 40: Transformation package construction phase

Goal: Obtain experience with the MDA approach

During the project we gained a lot of experience with the MDA approach. Not only did we implement a use-case using our MDA approach, we also developed our meta-models, models and transformations using an iterative development process. Because we used an iterative development process certain aspects of the MDA approach are different. For instance the development of our meta-models. We choose to built a part in the elaboration phase and a part in the construction phase. This caused problems when the meta-model changed to much. The models needed to be kept synchronized with the changed meta-model. We also gained experience with managing complexity of a MDA based development project. In general purpose programming language there are well known techniques to keep the source code maintainable. For instance the use of namespaces, packages and separation of concerns makes it easier to maintain and reuse code. In this project we gained some insights in how to manage complexity in MDA based development projects. For instance packaging QVT transformations and separating concerns in QVT.

Observations

During the development of UC2 we had to refactor the existing meta-models and restructure the model transformations. Because the menu structure was added and the master pages were introduced in this phase of the project the complexity of all the transformations increased. For building a system with general purpose programming language a build script can be used to control the building process. In our case the number of transformations increases and the ordering became important the need for a build script for transformations became evident. Such a script can be used to compose transformations into a building process. We will refer to this problem as "***No transformation composition***".

A second observation we made during this phase is that combining models is very hard at this moment. The user-experience models contain stereotypes that specify the type of action. In an ideal situation the implementation of an action should be modeled in a separate model. Once the user experience model is transformed, it is combined with the implementation of the specified action. We will refer to this problem as "***Composing models is difficult***". Composing models in QVT was difficult for two reasons. The first is reason is tool support. The tool we used has a QVT engine that is capable of handling multiple input models however, the user interface of the tool did not supported multiple

input models. The second reason is that a transformation language like QVT is focused on transforming model and not focused on merging models. QVT is to generic to define model compositions. (Kurtev & Didonet Del Fabro, A DSL for Definition of Model Composition Operators, 2006).

A detailed discussion of these issues can be found in section 5.3.

5.1.4 TRANSITION PHASE ITERATION 1

The transition phase is the only phase of the project that contained multiple iterations. In this section we evaluate the goals for the first iteration of the transition phase.

Goal: Deploy the system in a clean environment

In the first iteration of the transition phase the system was prepared for deployment in a clean environment. To make sure the system was installable and the installation manual was up to date we installed the system on a virtual machine with a fresh installation of Windows XP.

The installation on a separate machine showed that parts of the system are depended on XPO library and that this library must be installed on the host machine. Only delivering the XPO library in binary form seems inadequate.

Using a virtual machine with .NET 2.2 installed and a copy of IIS running we were able to demonstrate that our system compiled, ran and that is was possible to deploy the system.

5.1.5 TRANSITION PHASE ITERATION 2

The second transition phase was used to implement four selected changes. These changes are made to the product to find out what difficulties we could encounter if we need to make changes to a model driven created product. This section we evaluate the four change requests.

The following changes are evaluated:

CR1 A perfective change: Incoming and outgoing mail items lists should be paginated.

CR2 A corrective change: Aggregated input forms should have edit functionality.

CR3 A preventive change: A log file should be kept that logs the identifiers of objects, the time and the type of the object each time an object is changed.

CR4 An adaptive change: It should be possible to configure the database to be used in a configuration file.

We planned two weeks to implement the, at that time, unknown change requests. The change request were submitted by a senior application programmer. However, implementing the changes took two days. Much less than expected, this is what we changed to implement the changes:

Perfective change: Incoming and outgoing mail items lists should be paginated.

Pagination is build-in behavior of the .NET GridView component. To make pagination available in the ASP meta-model we had to add 2 attributes. A Boolean attribute to specify whether pagination must be used and an integer attribute to specify the number of rows per page. Two extra if-blocks were added to the Xpand template so the new options are written to the ASP files. The last thing that

needed to be done was the addition of a piece of QVT code. This QVT code sets the right properties in the ASP model if pagination is needed.

Corrective change: Aggregated input forms should have edit functionality.

The edit functionality is default available in the .NET GridView component, except when dynamic binding is used. In Arend MDA the datasource is dynamically bound to the gridview. To add the editor functionality to the gridview, three extra event handlers are needed. Two simple handlers that set/unset the current editable row when edit or cancel is clicked. The third event handler is slightly more complex. This event handler reads the changes and stores the new values. However, due to a bug in .NET reading the new values was hardly possible. Using a work-around the values could be read, but this meant adding a large block of code to each event handler. Therefore, the large part of this code was implemented as a library and a small block of code was added to each event handler. The addition of the event handlers also meant adding three attributes to the asp gridview tag in the ASP meta-model and the Xpand template for the ASP meta-model. The QVT transformation from user experience model to C# model was also modified. Three extra rules were added to create the event handlers in C#. A little QVT code was changed in the user experience to ASP transformation to set the event handlers in the ASP model.

A preventive change: A log file should be kept that logs the identifiers of objects, the time and the type of the object each time an object is changed.

The implementation of an audit trail meant changing the current implementation of the data layer. The C# domain model that was created from the UML model did not add any code for storing data. This is default behavior of the persistency library. If an object inherits XPObject, it will have a save method that stores the object. However, if we want to track changes in a log file we need to log all changes. And, we need transactions for this. We only want to write a record to the log if the transaction succeeded. If a transaction fails, nothing is changed and therefore no log record is needed. To implement CR3 we changed the save method for all persistent object to a save method with a transaction. The logging itself is implemented in a separated library. The QVT code that transforms the domain model into a C# model needed some changes, the save method is only added if the object inherits directly from XPObject and the save method implementation was added to the QVT code.

An adaptive change: It should be possible to configure the database to be used in a configuration file.

The configuration file for the database did not require any changes to the models or transformations. Inclusion of a special C# file containing one event handler and the code to set the database location variable for the persistency library was enough to implement it.

Observations

If changes are made to a product that is being developed with MDA techniques, some changes require modifications of the meta-models, models and transformations and some changes can be implemented separate. Our idea is that generic functionality should be implemented in the framework on which the application is build. Examples of such functionality are the editor and pagination. Other changes, that require extra code, should be implemented as a separate library. This improves reuse and makes it simpler to include it in the transformations and meta-models. The only code you add in the transformations should be the glue code that enables the application to use the library.

Changing the meta-models can be simple. Adding attributes to a model element is not a problem. Changing the name of an element or worse, removing an element resulted in our case in enormous problems, the models became unreadable for the tool. Therefore we think tool support is needed for model refactoring.

5.2 SUMMARIZED EVALUATION

The following table contains the goals for each of the iterations of the software development project. For each a goal a summary of the evaluation is given to provide a quick overview of the project evaluation. The names of the goals have been shortened to fit the table.

Evaluation of the goals			
Phase	Iter.	Goal	Summary
Inception	1	Determine scope	Scope was determined by selecting two use-cases from the original project.
		Determine architecture	The architecture was defined using the original project as reference architecture.
		Determine approach	Based on the reference architecture models, meta-models and transformations were determined.
		Setup development environment	Borland Together 2007 and Visual Studio 2005 were selected and installed as the development environment.
Elaboration	1	Stabilize architecture	The architecture as defined in the inception phase was used to implement the first use-case without any problems.
		Implement first use-case	An ASP and C# meta-model were created as PSM. The use-case was modeled at PIM level by a user-experience and a domain model.
		Deliver working system	The system was imported in Visual Studio to compile it. The debugger of Visual Studio was used for testing.
Construction	1	Implement complete system	The second use-case was implemented. To do so new meta-models were needed and existing meta-models were adapted. Extra user-experience diagrams modeled the second use-case, while the domain model was extended. A new structure was created to package the transformations.
		Obtain MDA experience	Extending existing meta-models and added new meta-models and transformations increased the complexity of the system. It showed some weaknesses in our approach and in the current tools and languages.
Transition	1	Deploy the system	The system was successfully deployed on a clean windows XP installation which acted as web server.
Transition	2	Add pagination to tables	Pagination is a feature supported by the .NET platform but was not supported by our models. Added a Boolean property to the ASP meta-model added support for pagination. Inserting an expression to the property in the transformation completed the implementation of the change request.
		Edit functionality for aggregated input forms.	This functionality was not available in the .NET platform in combination with our persistency library. Therefore we had to implement this function ourselves. The functionality was

			implemented as a library and on a number of places in the transformation. The user experience to C# transformation was adapted to work with the library.
		Log file	The log file was also implemented as a library or component. The domain to C# transformation was adapted to implement the new logging component. To make sure the log file reflects reality we also implemented transactions.
		Database configuration	A database configuration file was added using a .NET function. Including a special file in the project is enough to implement this change request.

We also provide a summary of the observations we made during this project. These observations are described in more detail in the next section.

Observations per phase			
Phase	Iter.	Observation	Summary
Inception	1	Selection of tools	In an MDA-based software development project selection of the right tools is important. Depending on the exact approach a selection must be made.
		Architecture of the approach	In an MDA-based software development project two architectures are created. An architecture for the system under development and an architecture for the tools to build that system.
Elaboration	1	Model and meta-model co evolution is difficult	All models must be adapted if the meta-model changes. Changes in the meta-models also need to be reflected in the transformations that use the meta-model.
		Structural incongruence increases transformation complexity	If the source and target meta-model are structural incongruent the transformation becomes more complex than a transformation from a source and target meta-model that are structural congruent.
		Lack of object orientation in QVT	Object orientation plays an important role in modern programming languages. MOF based meta-models use object oriented features like inheritance and abstract classes. These are however not supported by QVT.
Construction	1	Transformation composition	Building a complex system requires a number of transformations that work together to transform the models into a working system. To transform the system a build script is needed to execute the transformation in the proper order with the proper arguments.
		Model composition is hard	A development process like RUP incorporates the “4+1” view. This indicates that different models together model the system. The composition of this models is hard to describe using QVT operational mappings.
Transition	2	Implementation as library or component	Implementing new functionality in a MDA-based project forces the developer to create libraries or components that contain the functionality. The component as a whole is added to the project and the model transformations are adapted to include the component in the product.

5.3 DISCUSSION OF OBSERVED ISSUES

This section contains a discussion of the issues we observed during the project. For each of the issues one or more examples are given to provide a quick explanation of the issue. We also included more detailed description of the nature of the problems and provided possible solutions.

5.3.1 MODEL AND META-MODEL CO-EVOLUTION IS DIFFICULT

In our case study we used three meta-models, two UML 2.0 profiles and multiple instances of these meta-models. Changes made to the meta-model are not reflected in the instances of the meta-model. The same problem holds for the generated system once deployed. If the models change, the data currently stored in the production system does not reflect the changes in the system. A third problem is that transformations are depending on meta-models. In some cases if the meta-model changes, these changes ripple through the transformations.

The problem is that the model and meta-model do not co-evolve. The evolution of meta-models, models and also the transformations should be synchronized. Changes in the meta-model should be reflected by changes in the models and transformations. The problems of co-evolution are addressed in this paper (Wachsmuth, 2007)

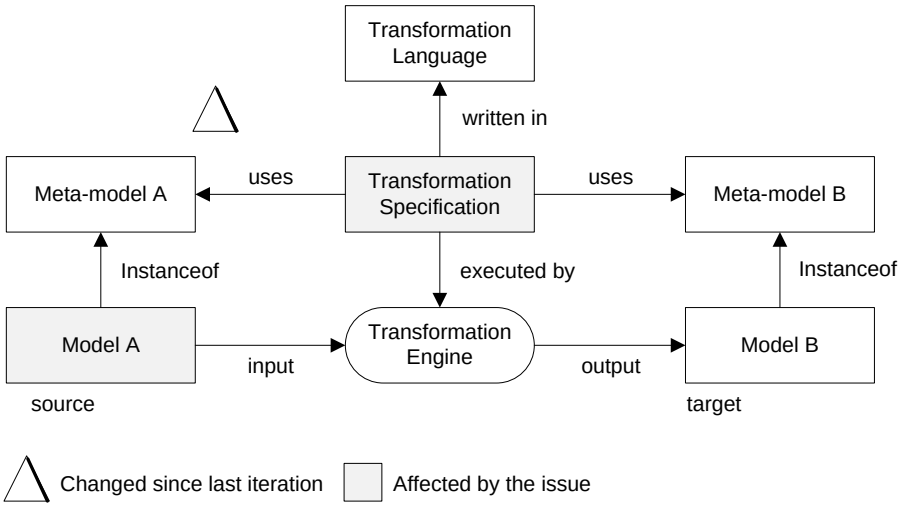


Figure 41: Transformation pattern with changes in meta-model A

Figure 41 illustrates the relation of this issue with the iterative approach. Since the meta-models used in the development of the system are developed in an iterative fashion this issue is a common problem. The delta in the figure illustrates that meta-model A has changed since the last iteration. This affects the models that are instances of meta-model A and the transformations based on this meta-model.

Refactoring is defined as “changing a system to improve its internal structure without altering its external behavior” (Fowler, Beck, Brant, Opdyke, & Roberts, 1999). The definition says nothing about what kind of system is improved, but in general the term refactoring is used in the context of source code. Changing source code without changing the behavior of the code. Nowadays many tools exist to help programmers refactor source code. Instead of modifying source code by hand, modern tools

understand the structure of programs and can change the code for the programmer, relieving him from the tedious task to locate and update all pieces of code that must be changed. To be sure that source code is still behaving correct after refactoring, the tool that helps refactoring must have knowledge about the code (Tichelaar, Ducasse, Demeyer, & Nierstrasz, 2000).

Refactoring tools can be build using a small number of primitive refactoring activities. These activities, called low-level refactorings, can be combined to form larger groups of refactorings, for instance to apply a design pattern, called high level refactorings. Examples of low-level refactorings are adding, renaming and deleting an element. All of these are supported by transformation languages like QVT (Wachsmuth, 2007).

Therefore we think that a solution to the problem of model and meta-model co-evolution might be model refactoring tools. If a meta-model is changed, the changes in the meta-model must propagate trough the transformations, models and further. Therefore, to enable model refactoring the system must know what models, transformations and other artifacts to refactor. This can be done by using a repository holding all artifacts.

Not all changes to models and meta-models are strictly refactorings. Evolution of models also includes creation and destruction of new model elements. Since creating/destructing new elements is not strictly refactoring, model refactoring might not be the proper term. However, both destruction and creation of new model elements can be seen as primitive activity and can also be described using transformations. We use the term model refactoring because programmers have are used to this term.

5.3.2 STRUCTURAL INCONGRUENCE INCREASES TRANSFORMATION COMPLEXITY

One of the key issues we found while implementing model transformations was that the complexity of the transformation increases if the source meta-model is not structural congruent with the target meta-model. The more incongruent the meta-models are the harder it is to bridge the structural gap. This problem arises for instance when a transformation from a ‘flat’ model to a very ‘hierarchical’ model is written. With a ‘flat’ model we mean models with a meta-model that does not contain many nested element; with a ‘hierarchical’ model we mean a model with a meta-model that does contain much nested elements. An example of a ‘flat’ model is a UML class diagram. A class diagram contains packages, packages contain classes and classes contain attributes. Attributes do not contain other elements. In this example we can go three steps deep. An example of a ‘hierarchical’ meta-model is the ASP meta-model we created. To display a table with a title on a page one needs many tag elements. A page element, a body element, a form element, a panel element, a bold element, a text element, a table element etc. This meta-model is more hierarchical from structure and in this case we can go up to seven steps or more deep.

Bridging the structural gap between two meta-models requires extra rules in the transformation step that increase the complexity of the transformation. The following QVT code sample illustrates this:

```
1  mapping transformToPanel( screen: uml20::classes::Class): aspx::ASPPanelTag
2  {
3       init {
4               --Generate the table with a row for each property
5               var table : aspx::TableTag := generateTable('','table');
6               table.tags+= screen.ownedAttributes->
7                               collect( prop | property2field(prop) )->asOrderedSet();
8       }
9       runat:=aspx::runatEnumeration::server;
10      id:=toPanelId('fields');
11      tags+=table;
12 }
```

```
13 mapping transformToCells ( prop : uml20::kernel::Property) : aspx::TableRowTag
14 {
15     init {
16             var cell1 : aspx::TableColumnTag := tableCell();
17             var cell2 : aspx::TableColumnTag := tableCell();
18             cell1.tags+=generateCData( toNiceLabel(prop.name));
19             cell2.tags+=aspLabel( prop.name );
20             cell2.tags+=generateComment(prop.getPropertyValue('field'));
21     }
22     tags+= cell1;
23     tags+= cell2;
24 }
```

Code section 7: Hierarchical complexity in QVT

In Code section 7, a class and the properties it contains are mapped to a asp panel containing a table (line 5), containing rows (line 13), containing cells (line 16,17) and the label and text (line 19,20). The source structure has depth two, while the target structure has depth four. In this example this is solved in the init sections of the mappings. The extra model elements are created and the output of the mappings is added to these model elements.

Code section 8 shows the same transformation as described in Code section 7, but without increasing the complexity of the mappings. Instead of more complex mappings, more mappings are needed to bridge the structural gap. Many very simple mappings are introduced just to create all the model elements needed.

```
1  mapping transformToFields( screen: uml20::classes::Class ) : aspx::ASPPanelTag
2  {
3      runat:=aspx::runatEnumeration::server;
4      id:=toPanelId('fields');
5      tags+=transformToFieldsTable( screen );
6  }
7
8  mapping transformToFieldsTable( screen: uml20::classes::Class ) : aspx::TableTag
9  {
10     class:= 'ta ble';
11     tags+=screen.ownedAttributes->collect( prop | transformToFieldRows(prop) );
12  }
13
14 mapping transformToFieldRows( prop : uml20::kernel::Property ) : aspx::TableRowTag
15 {
16        tags+=transformToTextCell( prop );
17        tags+=transformToLabelCell( prop );
18 }
19
20 mapping transformToTextCell( prop : uml20::kernel::Property ) : aspx::TableCellTag
21 {
22        tags+=generateCData( toNiceLabel(prop.name) );
23 }
24
25 mapping transformToLabelCell( prop : uml20::kernel::Property ) : aspx::TableCellTag
26 {
27        tags+=aspLabel( prop );
28 }
29
```

Code section 8: QVT increased number of mappings

The code in Code section 8 looks readable and maintainable code at first sight. But, instead of two mapping rules it now contains five mapping rules. This is not a problem for such a small example, but this may become problematic for larger transformations.

5.3.3 LACK OF OBJECT ORIENTATION IN QVT

Currently the specification of QVT contains no polymorphism for mappings. This polymorphism is very useful in the case that properties of the source element determine the type of the target element. Consider the following example in which a UML property is mapped to an ASP input. Depending on the type field of this UML property it maps it to a ASP text input or a ASP checkbox. The QVT code to implement this mapping would look like:

```
1  query transformToInput( property : uml20::kernel::Property ):aspx::AbstractInputTag
2  {
3       if property.type.name = 'String' then object aspx::TextInputTag { … } else
4       if property.type.name = 'Boolean' then object aspx::CheckboxTag { … } else
5       undefined endif endif;
6  }
```

Code section 9: Polymorphism in QVT

The problem is that Code section 9 is not valid according to the QVT engine of Borland Together. Both types may be descendants of the AbstractInputTag, but the QVT engine considers them different types and a mapping that may return two different types is not allowed.

According to the specification of QVT it is possible to implement the transformations as specified in Code section 9. With the use of disjunct mapping operations. With a disjunct mapping operations we can define an ordered list of mappings with guards. The engine loops trough the list of mappings to find the first mapping operation that is available. Using this mechanism we could implement the example:

```
1  mapping transformToInput( property: uml20::kernel::Property )  :
2  aspx::AbstractInputTag disjuncts transformToText, transformToCheckbox(){}
3
4  mapping transformToText( property: uml20::kernel::Property ) :aspx::TextInputTag
5  when{ property.type.name = 'String'}
6  {
7    …
8  }

9  mapping transformToCheckbox( property: uml20::kernel::Property ) :aspx::CheckboxTag
10 when{ property.type.name = 'Boolean'}
11 {
12   …
13 }
```

Code section 10: Hardcoded polymorphism in QVT

Code section 10 illustrates how the mechanism can be implemented using disjunct mappings. If the transformToInput mapping is called, the engine looks for the first mapping that is available. Depending on the type the transformToText or the transformToCheckbox becomes available. Other types can be supported by adding transformations to the list. This mechanism is quite nice but does require more attention than a single mapping that has an abstract return value. If other types are added to the model, the list of disjunct mappings must be updated to support the new type. Disjunct mappings are in the QVT specification but not supported by the QVT engine of Borland Together 2007.

In general purpose programming languages it is a good practice to use object-oriented mechanisms to cope with complexity. In eCore meta-models it is possible to specify inheritance for model elements. In our case study the ASP meta-model was based on the principles of specialization. The core model element defined in the ASP meta-model is the abstract tag. This model element is forms the basis for all tags used in the ASP meta-model. All other tags are specializations of this one model element. The

idea is that tags can always be nested in tags. This mechanism is built in the abstract tag. It contains a set of other abstract tags to support nesting.

In the QVT transformations the support for OO mechanisms in the meta-models like specialization of model elements is poor. Consider the following example in which we would like to map one source element into a sequence of target elements. Each target element of the same super type, but different sub types. For instance, a UML class is mapped to a textual title and a html table. The obvious QVT code would look like:

```
1  query transformClassToTitledTable( table : uml20::classes::Class ) : Set(aspx::Tag)
2  {
3       Set {
4              object aspx::CDataTag{ data:='Table: '+table.name;},
5              object aspx::TableTag{ class:='table'; }
6       }
7  }
```

Code section 11: Poor inheritance in QVT

The idea behind Code section 11 is that the query returns a set of tags. Each tags contains such a set of abstract tags. Using the '+=' operator it is possible to add elements and sets of elements to this set. However, the construction as shown in Code section 11 is not allowed. The QVT engine complains that the CDataTag and the TableTag are not of the same type. The QVT engine is wrong, they both descent from Tag.

Object oriented mechanisms like inheritance can be used to support software evolution. In this case, the QVT language does not support the mechanisms to implement maintainable mappings. If the meta-models change all mappings must be adapted. Even simple changes like the introduction of a new specialized type lead to changes in the transformations. Figure 42 illustrates the relation between the iterative development process we used to develop our system and the issues.

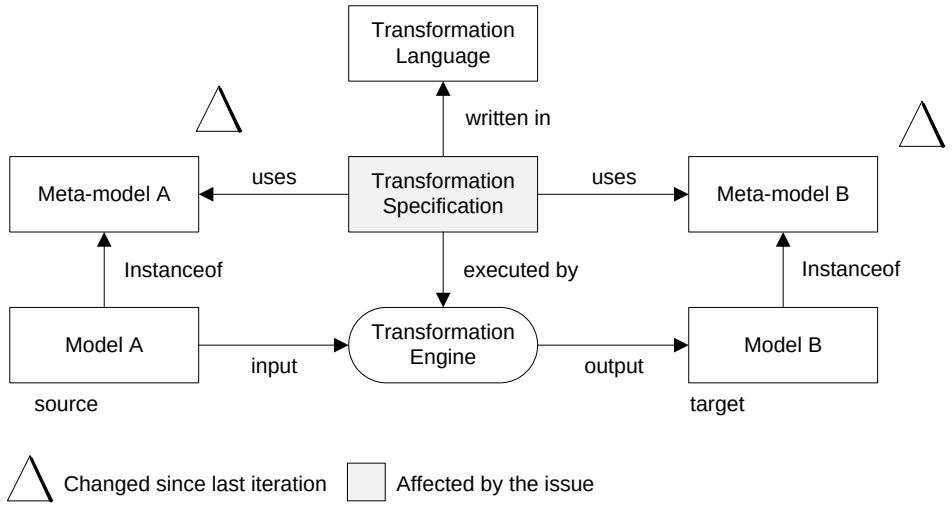


Figure 42: Transformation pattern for poor inheritance issue

5.3.4 TRANSFORMATION COMPOSITION

If model transformations are used in the development of a software system the number of transformations involved grows with the size of the system. In our case study we had a very small system with a limited domain, but still seven transformations working together to transform the

source models into a finished product. The composition of these transformations becomes increasingly complex. The ordering, interaction and execution parameters all need to be controlled to make sure the overall transformation from models to product is successful.

Automation of the transformations becomes necessary when the complexity of all transformations becomes hard to manage. For general purpose programming languages build scripts are used to handle the transformation from source code to binary code. These build script have become more powerful and sometimes have become DSL's for specifying the build process. For model transformations such languages do not exist yet. Ant script can be used to drive the transformation engine, but these Ant scripts lack the semantics of the QVT code. Therefore Ant cannot make decisions based on the results of a transformation. Ant is not capable of reading the models.

The relation to the iterative development process for this issue is that execution of the transformations is a repeating process. After each iteration the transformations must be executed to deliver a working system. In Figure 43 the transformation specification is both affected and changed. This is because the lack of transformation composition can lead to a hard to maintain transformation process.

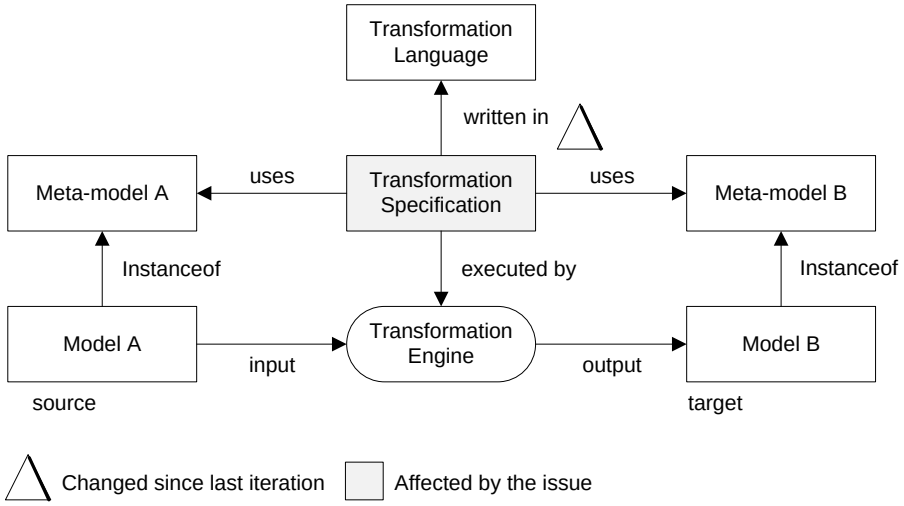


Figure 43: Pattern for the issue of transformation composition

The QVT specification (Object Management Group, 2007) contains a short section about composing transformations. In the section is explained how transformations as a whole can be extended and how other transformations can be invoked. The following example demonstrates how an existing transformation can be extended by first calling another transformation and then executing the inherited transformation:

```
1  transformation CompleteUml2Rdbms(in uml:UML,out rdbms:RDBMS)
2    access transformation UmlCleaning(inout UML),
3    extends transformation Uml2Rdbms(in UML,out RDBMS);
4  main() {
5      var tmp: UML = uml.copy();
6      var retcode := (new UmlCleaning(tmp))->transform(); // performs the "cleaning"
7      if  (not retcode.failed())
8           uml.objectsOfType(Package)->map packageToSchema()
9      else raise "UmlModelTransformationFailed";
10 }
```

Code section 12: Transformation composition (Object Management Group, 2007)

The high level constructions as described in the specification allow the programmer to specify a global transformation that coordinates the execution of all other QVT transformations. One problem is that QVT itself is limited to QVT. In practice Ant should still be used to perform other steps of the development process like invoking the model to text transformations and the build process.

Currently this part of the specification is not implemented in Borland Together 2007.

5.3.5 COMPOSING MODELS IS DIFFICULT

In our case study the user experience models contained stereotypes that indicated the type of action for operations on a screen class. The semantics of the stereotypes is defined in the transformation that transforms the user experience model into a C# model. However, the semantics of the operations are hard coded in the transformation and not very easy to extend. To make this more easy to extend we thought of implementing the template for the actions as a separate model. One model corresponds with one stereotype in the user experience model. For instance, a UML sequence diagram can be used to model actions or a UML state chart. However, combining the model of the action with another model with QVT Operational Mappings is very difficult.

We think there are three possible methods to use models as a template. One idea is that transformation languages should enable the programmer to invoke the execution of another transformation and supply the parameters. The output of that transformation can then be processed in the running transformation.

An example of how this hypothetically could be implemented in QVT:

```
1  mapping transformToAction( op : uml20::kernel::Operation ) : CSharp2::CSMethod
2  {
3       name:='Operation';
4       implementation:=invoke( 'createOperation.qvt' , 'toAction', 'param'  );
5  }
```

Code section 13: QVT Invoke statement

The idea of the above QVT code is that the *invoke* statement can be used to invoke other transformations and obtain the results. In this example three parameters are given. The name of the transformations, the mapping rule to call and the parameters for this mapping rule. If this hypothetical *invoke* statement would exist it would be possible to model actions as a sequence diagram and a transformation that combines the parameters and the sequence diagram to implement the action.

A second method is to use a model composition language. A model composition language can be used to describe how models are related and how models can be merged. The primary element of a model composition language is the composition operator. A composition operator describes how two or more models can be combined into a new model. Composition operators can be described in general purpose transformation languages or a domain specific language can be used (Kurtev & Didonet Del Fabro, A DSL for Definition of Model Composition Operators, 2006).

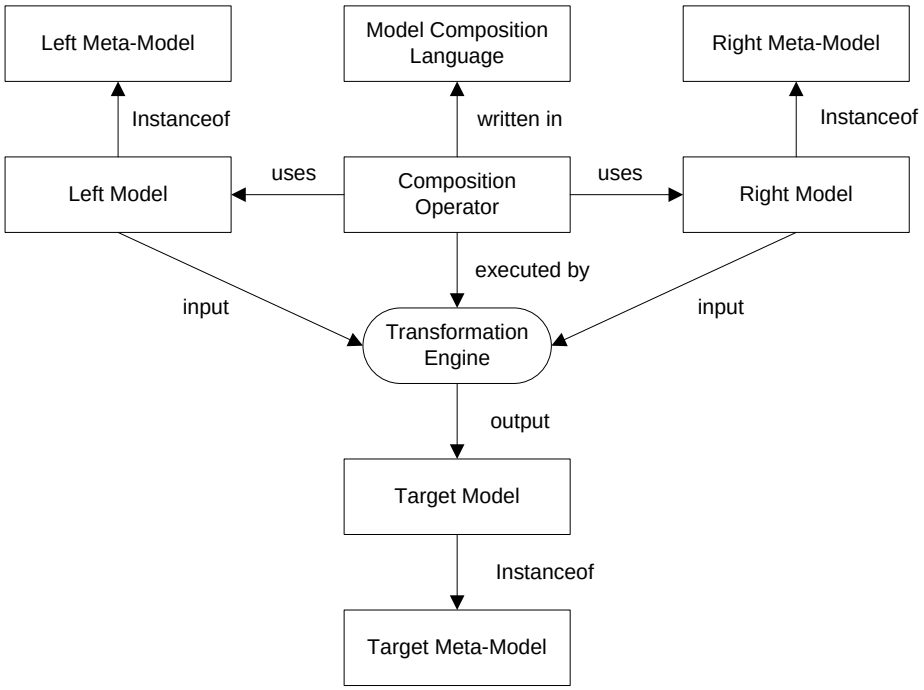


Figure 44: Model composition pattern

The last idea is to use model weaving to weave in the actions. Weaving is a term related to Aspect Oriented Programming (AOP). In AOP a system can be described from different views using aspects. Using an aspect language a programmer can specify how a certain aspect can be woven in to an application. For instance, a system is build with very specific security needs. These needs can be specified as a separate concern. The system itself is implemented without security needs, but an aspect weaver is able to weave in the security aspect to add the needs to the system. Current AOP solutions like AspectJ (Kiczales, Hilsdale, Hugunin, Kersten, Palm, & Griswold, 2001) or Compose* (Bergmans) focus on general purpose programming languages. Model weavers apply the AOP principles on model level. The Atlas Model Weaver is one system that can do this (Didonet Del Fabro, Bézivin, & Valduriez, 2006).

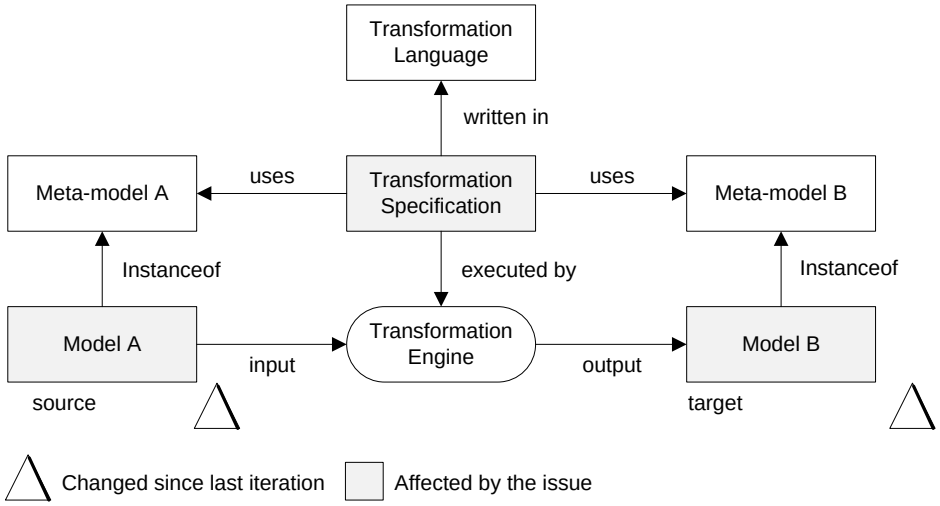


Figure 45:Pattern for the issue of model composition

The issues with model composition in general can be related to the iterative development process in two ways. RUP, the development process we used in this project is based on the “4+1” view. The system is modeled from different views in different models instead. Each model has a specific concern to model. The second relation between this issue and iterative development processes is that a the implementation of a system is spread over multiple iterations. An example is security. In one iteration the basic system is constructed, while in another iteration the security aspects are added. In model driven engineering this can be done using model composition. Figure 45 provides an illustration of the relation between the issue of model composition and iterative development process.

5.4 CONCLUSION

In this chapter we evaluated the project and discussed the issues we observed. For each of the phases of the project we listed the goals and evaluation per goal. At the end of each phase we also described our observations. Using these observations we capture our operational knowledge of iterative MDA-based software development. For instance, the iterative aspect of this project showed us that models need to be adapted when meta-models change. In this project the meta-models changed because we implemented only that part of the meta-model that we needed. In the next iteration the meta-models was adapted to meet the new use-case. Changing meta-models is however not a simple operation. Removing properties from our meta-models resulted in errors when opening models that were still based on the previous version of the meta-model.

For each of the issues that we observed during the project we described the issue, we provided examples to illustrate the problem and we even provided solutions for the problems if there was a known solution.

Chapter 6

6 CONCLUSIONS AND FUTURE WORK

In this chapter we present our results and conclusions. Based on the conclusions we also have some recommendations for future work.

6.1 PROJECT SUMMARY

This study began with two questions. We wanted to know what the critical issues were if MDA-based software development is applied in an iterative development process and we also wanted to know what the issues were with respect to maintenance activities and MDA. To answer the research questions we first defined MDE in general, MDA and the basic concepts we used throughout this thesis. Besides the theories behind model driven engineering we also introduced iterative software development and two examples of an iterative development process.

Besides the research questions there was a more general problem that resulted in this study. Both the University of Twente and Getronics PinkRoccade wanted more operational knowledge about MDA. Therefore we chose to conduct a case study to both answer our research questions as to obtain operational knowledge about how MDA can be applied in an iterative development process.

During the case study we built a small sized application. This application was based on an existing application developed at Getronics PinkRoccade. Therefore requirements and a good reference architecture were available to us which saved time. For the case study we used the Rational Unified Process as our software development process. Our project contained four phases. In the inception phase, an architecture for the application was created and we also created an architecture for our MDA approach. We defined the models to use and how the transformations would transform the source models into a working application. We used the elaboration phase to test our approach and to build the major part of the meta-models, transformations and models. As we expected, building the meta-models and transformations requires a larger part of the time. Therefore we selected only a small use-case to implement in this phase. The meta-models, transformations and models were not complete, but complete enough to test our approach and to implement the single use-case. This worked very well. During the construction phase a larger and more complex use-case was selected and implemented. The meta-models, transformations and models were extended to support the selected use-case.

6.2 ITERATIVE MDA-BASED DEVELOPMENT

Applying MDA-based software development in an iterative development process is possible. We did not encounter any critical issues, issues that halted our progress. However some caution is needed. In our project we developed three meta-models and seven transformations just to implement two use-cases. We recognized some problems with our approach: model and meta-model co-evolution, structural incongruence increases transformation complexity, lack of object-orientation in QVT, no transformation composition and composition of models is difficult. We expect that developing a more complex application requires model driven techniques that are currently still a topic of research. For instance model composition and transformation composition are needed to cope with the complexity of combining and transforming multiple models into an application. Model evolution tools are needed to cope with changes in the future. Changing meta-models requires too much time without proper tooling (van Deursen, Visser, & Warmer, 2007).

One major problem that people expect when applying MDA is that development of meta-models for applications and specific problem domains is very time consuming and that this does not fit in the iterative development process. We showed that it is possible to develop the meta-models needed. We created an architecture for our approach in the inception phase and implemented the meta-models, models and transformations in the elaboration and construction phase. We think that developing the meta-models needed can be part of the project. Specific meta-models are developed in the project itself and more general meta-models can be reused or bought. The same holds for transformations. Currently a large number of meta-models and model transformations is available on the internet (The Atlantic Zoo) (ATL Transformations).

Modern software development processes provide mechanisms to tailor the process. Both RUP and OpenUP provide tooling to generate a customized and fully documented software development process. These tools can be used to develop a process that incorporates MDA. Some work has already been done on this topic (Brown & Conallen, 2005) (Eclipse Process Framework OpenUP/MDD, 2006), but there is currently not a software development process for MDA.

6.3 MDA AND MAINTENANCE

During the maintenance phase of this project we did not encounter critical issues. We can conclude that in this project the implementation of the change request finished earlier than expected and that we did not encounter critical issues. We did however make a number of observations that are related to some of the problems we encountered during the construction of the system. The first observation we made was that new functionality can be best implemented as a component. The model transformations can be adapted to incorporate the new component in the system. We illustrated this by implementing the log functionality and the editor for the aggregated input forms. A second observation we made is that maintenance may require models to be changed or even meta-models. Changing meta-models is currently a problems and the change made to a meta-model propagate trough the system. If an element changes, the models and transformations based on the meta-model must also change. We described this problem as “model and meta-model co-evolution is difficult”.

The overall conclusion with respect to maintenance and MDA is that maintenance can be applied at another level. Instead of adapting multiple lines of code scattered through the system we were able to modify the transformations to include the change. Running al the model transformations was enough to adapt the system to the change request. A second advantage of the MDA approach is that it forces the use of components. The disadvantage of MDA with respect to maintenance is that the

problem of model evolution is currently not solved by tools. At least, Borland Together 2007 does not provide the functionality to keep meta-models, models and transformations in sync.

6.4 IMMATURITY OF TOOLS

An important aspect of MDA is that it heavily relies on tools. During the case study it became clear that the tools we selected contained all the features we needed for our study. However, a problem was that many of the features were undocumented, contained bugs or didn't work at all. The tool vendor worked hard to support us with our problems and promised to solve some of the problems in a newer version. In the end all features we needed were working or work-around's were available, but we lost valuable time.

Another indication that tools are not mature enough is that QVT transformations became complex and harder to maintain. The QVT engine shipped with our tool did not implement the full QVT specification. It was clear that the tool focuses on implementing the basics of QVT and therefore supports only simple transformations. This did not introduce any critical problems to our case study, but showed us that execution and maintenance of QVT transformations becomes more complex as the size of the project grows. In the QVT specification some features are specified to cope with this growing complexity.

We think that the current state of the practice is that MDA is certainly possible and that MDA can be applied in combination with an iterative development process. We can conclude that current tools and languages support the basic features needed to apply MDA, but that problems can be expected when complexity of the models and transformations increases. The last conclusion, with respect to maintenance is that maintaining models instead of source code is easier. Changing a single property results in changes all over the source code. Evolution of MDA-based systems is still a problem.

6.5 FUTURE WORK

In this section we propose some topics that need further attention. We think providing solutions to the following problems will increase the productivity of MDA and the acceptance by the software industry.

6.5.1 MODEL COMPOSITION

Modern software system cannot be specified in a single model. A modeling language like UML uses the "4+1" view model (Kruchten, Architectural Blueprints—The "4+1" View Model of Software Architecture, 1995). This indicates the multiple models are used to describe a single system. In our case study we encountered the problem that we would like to be able to specify actions in separate models (section 5.3.5). These actions should than be composed on the model of the system to form a complete model of the system.

Current transformation languages have very limited support to compose models. We think that more research on this topic is necessary to cope with the complexity of today's software systems. Some work has been done to easy model composition: (Baudry, Fleurey, France, & Reddy, 2005), (Pastor, 2006).

6.5.2 TRANSFORMATION COMPOSITION

One of the issues we encountered during our project was that the composition of the transformations cannot be specified in Borland Together 2007. The QVT specification (Object Management Group, 2007) contains a short section about how transformations may be specified on a higher level.

We think that in an ideal situation there should be multiple levels of abstraction in a transformation language. A low level transformation language for model elements and a high level transformation language for models. This would allow transformation programmers to specify the transformation process like build scripts can specify the build process for general purpose programming languages. In general we call for a language that allows us to specify the workflow of the transformation process.

6.5.3 MODEL META-MODEL CO-EVOLUTION

Modern integrated development environments like the Eclipse Platform provide refactoring tools to change the structure of the code without changing behavior nor breaking the code. Refactoring is based on small refactoring steps called refactorings. It should be possible to implement refactoring tools for models based on the principle of these small refactorings. Refactoring meta-models may be even more interesting. The transformations for the instances of a meta-model may be generated based on the refactorings. The techniques that were originally developed for general programming languages should be adapted to models. It should even be possible to extend the mechanism to not only support refactoring, but also construction and destruction of model elements. These tools should be able to provide automated support for model and meta-model co-evolution. Some work on this field has already been done: (Tichelaar, Ducasse, Demeyer, & Nierstrasz, 2000) (Wachsmuth, 2007).

6.5.4 MODEL DRIVEN DEVELOPMENT PROCESS

Currently there are is no software development process that is tailored to model driven development. MDA and MDE in general can fit in an iterative development process but do require some changes to the process. Some of these changes are described in this thesis (Section 3.4.4 & 3.5.4) , but do not provide a complete model driven development process. We think that OpenUP or RUP can be extended with activities, roles and artifacts to support model driven development.

Bibliography

Aldawud, O., Elrad, T., & Bader, A. (2003). Uml Profile For Aspect-Oriented Software Development. *Proceedings of Third International Workshop on Aspect-Oriented Modeling.* Boston, USA.

Allilaire, F., Bézivin, J., Jouault, F., & Kurtev, I. (2006). ATL: Eclipse Support for Model Transformation. *Eclipse Technology eXchange workshop.* Nantes, France.

Amber, S. W. (2007, 3 3). *History of the Unified Process*. (Ambysoft) Retrieved 12 10, 2007, from Enterprise Unified Process: http://www.enterpriseunifiedprocess.com/essays/history.html

Ambler, S. W. (2007, July 15). *Agile Software Development: Definition*. Retrieved April 2, 2008, from Agile Modeling: http://www.agilemodeling.com/essays/agileSoftwareDevelopment.htm

American Institute of Aeronautics & Astronautics. (1998). *AIAA Guide for the Verification and Validation of Computational Fluid Dynamics Simulations.* American Institute of Aeronautics & Astronautics.

Arnold, D. (2004). *C# Compiler Extension to Support the Object Constraint Language Version 2.0.* Ontario, USA: Carleton University.

ATL Transformations. (n.d.). *ATL Transformations*. (The Eclipse Foundation) Retrieved March 28, 2008, from Eclipse Home: http://www.eclipse.org/m2m/atl/atlTransformations/

Balduino, R. (2007, August). Introduction to OpenUP.

Basili, V. R., & Larman, C. (2003). Iterative and Incremental Development: A brief history. *Computer , 36* (6), 47-56.

Baudry, B., Fleurey, F., France, R., & Reddy, R. (2005). Exploring the Relationship between Model Composition and Model Transformation. *Aspect Oriented Modeling (AOM) Workshop.* Montego Bay, Jamaica.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., et al. (2001). *Manifesto for Agile Software Development*. Retrieved April 2, 2008, from Agile Manifesto: http://agilemanifesto.org/

Bennett, K., & Rajlich, V. T. (2000). Software Maintenance and Evolution: a Roadmap. *Proceedings of the Conference on The Future of Software Engineering*, (pp. 73 - 87). Limerick, Ireland.

Bergmans, L. (n.d.). *ComposeStar*. (University of Twente) Retrieved March 13, 2008, from ComposeStar: http://janus.cs.utwente.nl:8000/twiki/bin/view/Composer/WebHome

Bézivin, J., & Gerbé, O. (2001). Towards a precise definition of the OMG/MDA framework. *Proceedings of the 16th IEEE international conference on Automated software engineering*, (p. 273). USA.

Brown, A., & Conallen, J. (2005, May). *An introduction to model-driven architecture*. (IBM) Retrieved 12 6, 2007, from IBM Developerworks: http://www.ibm.com/developerworks/rational/library/may05/brown/index.html

Buschman, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-Oriented Software Architecture: A System Of Patterns.* West Sussex: Wiley.

Chapin et al. (2000). Types of Software Evolution and Software Maintenance. *Journal of Software Maintenance and Evolution: Research and Practice* (13), 3-30.

Chauvel, F., & Fleurey, F. (2007). *Kermeta Language Overview*. Retrieved April 02, 2008, from http://www.kermeta.org

Czarnecki, K., & Helsen, S. (2003). Classification of Model Transformation Approaches. *Proceedings of the OOPSLA'03 Workshop on the Generative Techniques in the Context Of Model-Driven Architecture.* Anaheim, USA: ACM.

Deursen, A. v., Klint, P., & Visser, J. (2000). Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices , 35* (6), 26-36.

Didonet Del Fabro, M., Bézivin, J., & Valduriez, P. (2006). Weaving Models with the Eclipse AMW plugin. *Eclipse Summit Europe.* Nantes, France: University of Nantes.

Eclipse Foundation. (n.d.). *Eclipse Process Framework Project*. Retrieved April 2, 2008, from Eclipse : http://www.eclipse.org/epf/

Eclipse Process Framework OpenUP/MDD. (2006, October 3). Retrieved April 2, 2008, from Eclipse: https://bugs.eclipse.org/bugs/show_bug.cgi?id=138867

Elrad, T., Filman, R. ,., & Bader, A. (2001). Aspect oriented programming. *Communications of the ACM , 44* (10).

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code,.* Addison-Wesley.

Fuentes-Fernández, L., & Vallecillo-Moreno, A. (2004, April). An Introduction to UML Profiles. *Upgrade , V* (2), pp. 6-13.

Gardner, T., Griffin, C., Koehler, J., & Hauser, R. (2003). *A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard.* Object Management Group.

Glenn Brookshear, J. (2000). *Computer Science an Overview.* Addison Wesley.

IBM. (2003). Rational Unified Process.

IEEE. (1990). *IEEE Standard Glossary of Software Engineering Terminology.* New York: IEEE.

Kent, S. (2002). Model Driven Engineering. *Lecture Notes In Computer Science , 2335*, 286 - 298.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., & Griswold, W. G. (2001). An overview of AspectJ. *ECOOP.* Budapest: Springer.

Kleppe, A., Warmer, J., & Bast, W. (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison Wesley Professional.

Kozaczynski, W., & Thario, J. (2002). Transforming User Experience Model To Presentation Layer Implementations. *OOPSLA 2002.*

Kruchten, P. (1995). Architectural Blueprints—The “4+1” View Model of Software Architecture. *IEEE Software , 12* (6), 42-50.

Kruchten, P. (2000). *The Rational Unified Process: An Introduction* (Vol. 2). Addison Wesley Professional.

Kurtev, I. (2005). *Adaptability of Model Transformations PhD Thesis.* Enschede: University of Twente.

Kurtev, I., & Didonet Del Fabro, M. (2006). A DSL for Definition of Model Composition Operators. *Lecture Notes in Computer Science , 4379*, 21-25.

Kurtev, I., & van den Berg, K. (2005). Building Adaptable and Reusable XML Applications with Model Transformations. *Proceedings of the 14th international conference on World Wide Web* (pp. 160 - 169). Chiba, Japan: IW3C2.

Kurtev, I., Bézivin, J., Jouault, F., & Valduriez, P. (2006). Model-Based DSL Frameworks. *OOPSLA* (pp. 602 - 616). New York, USA: ACM.

Lawley, M. J. (n.d.). Retrieved March 10, 2008, from Tefkat - The EMF Transformation Engine.: http://sourceforge.net/projects/tefkat

Lyons, B. (2007). *The Open Unified Process*. (Number Six Software) Retrieved February 27, 2008, from Numbersix: http://www.numbersix.com/news/n6articles/openUp.html

Mariam Webster. (n.d.). *Mariam Webster Online Dictionary*. Retrieved September 26, 2007, from Mariam Webster: http://www.m-w.com/

Mens, T., Czarnecki, K., & Gorp, P. v. (2005). Discussion -- A Taxonomy of Model Transformations. *Dagstuhl Seminar on Language Engineering for Model-Driven Software.* Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI).

Mernik, M., Heering, J., & Sloane, A. M. (2005). When and How to Develop Domain-Specific Languages. *ACM Computing Surveys , 37* (4), pp. 316 - 344.

Object Management Group. (2003). *Common Warehouse Metamodel (CWM) Specification.* Object Management Group.

Object Management Group. (2003). *MDA Guide version 1.0.1.*

Object Management Group. (2006). *Meta Object Facility (MOF) Core Specification.*

Object Management Group. (2007). *MOF 2.0/XMI Mapping, Version 2.1.1.*

Object Management Group. (2007). *MOF QVT Final Adopted Specification.*

Object Management Group. (2005). *Software Process Engineering Metamodel.*

Object Management Group. (2003). *UML 2.0 OCL Specification.* Object Management Group.

Object Management Group. (2007). *Unified Modeling Language: Superstructure.*

Oever, D. v., & Vos, G. (2007). *Aspect Oriented User Restriction Language.*

Pastor, R. (2006). *Model Composition: Definition of Model Composition Properties.* University of York.

Pfleeger, S. L., & Atlee, J. M. (2005). *Software Engineering.* Prentice Hall.

Project Quadread. (2008). Retrieved April 03, 2008, from Quadread : http://quadread.ewi.utwente.nl/

Rational Corporation. (1998). *Rational Unified Process: Best Practices for Software Development Teams*. Retrieved April 3, 2008, from IBM Developerworks: http://www.ibm.com/developerworks/rational/library/253.html

Rational. (2004). *Rational® UML Profile for Business Modeling*. Retrieved April 3, 2008, from IBM Developer works: http://www.ibm.com/developerworks/rational/library/5167.html

Royce, D. W. (1987). Managing the development of large software systems: concepts and techniques. *International Conference on Software Engineering* (pp. 328 - 338). Monterey, California, United States: IEEE.

Rumbauch, J., Jacobson, I., & Booch, G. (1999). *The Unified Modeling Language Reference Manual.* Addison-Wesley.

Seidewitz, E. (2003). What Models Mean. *IEEE Software , 20* (5), 26-32.

Seifert, T., & Beneken, G. (2005). Evolution and Maintenance of MDA applications. In S. Beydeda, M. Book, & V. Gruhn, *Model-Driven Software Development* (pp. 269-286). Berlin Heidelberg: Springer.

The Atlantic Zoo. (n.d.). *The Atlantic Zoo*. (The Eclipse Foundation) Retrieved March 28, 2008, from Eclipse Project: http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/

Tichelaar, S., Ducasse, S., Demeyer, S., & Nierstrasz, O. (2000). A Meta-model for Language-Independent Refactoring. *ISPSE* (pp. 154-164). IEEE.

van Deursen, A., Visser, E., & Warmer, J. (2007). Model-Driven Software Evolution: A Research Agenda. *Proceedings 1st International Workshop on Model-Driven Software Evolution* (pp. 41-49). Delft: Technical University of Delft.

Visser, E. (2004). Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9. *Lecture Notes in Computer Science , 3016* (June).

Wachsmuth, G. (2007). Metamodel Adaption and Model Co-adaption. (E. Ernst, Ed.) *Lecture Notes in Computer Science* (4609), pp. 600-624.

Wegener, H. (2002). Agility in Model-Driven Software Development Implications for Organization, Process, and Architecture. *OOPSLA.* ACM.

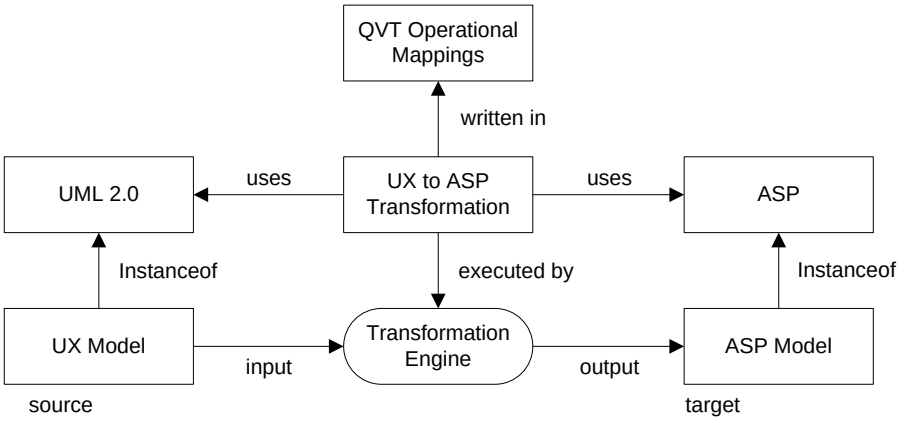
APPENDIX A PROBLEM ANALYSIS RESULTS

The last appendix contains the results of the problem analysis session we held during at the beginning of this project. The list contains a number of possible causes for the following problem: “MDA is not applied on a large scale”. All stakeholders of this project were asked to come up with possible causes. After the first session the possible causes were send to the stakeholders and we asked them to rank each of the possible causes. The problems with the highest rank were selected as a starting point for this project. The following list contains the results of the problem analysis sessions:

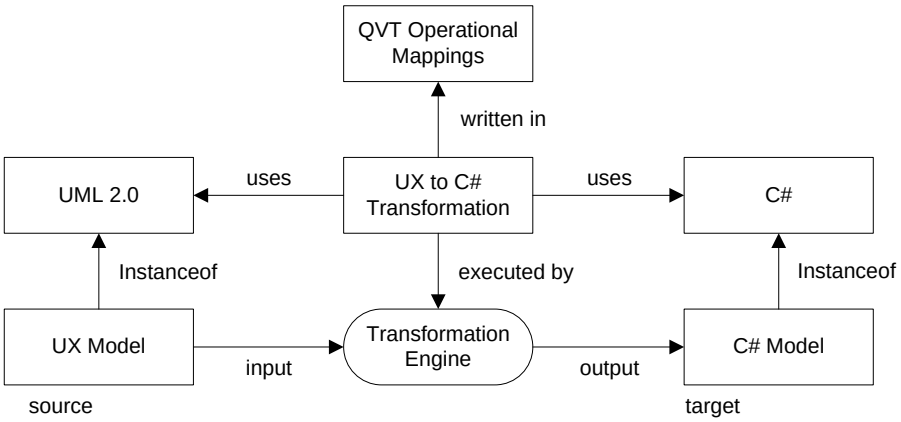
	Score
MDA is not iterative	**9**
Lack of experience with MDA and maintenance	**8**
It is unclear whether models are easier to maintain than source code	**5**
Can we use MDA for prototyping?	5
Tools cannot model dynamic behavior	4
There are lots of solutions, but a lack of integration	4
It is unclear how different models can be kept consistent	3
Unclear whether tools support debugging at model level	2
Special personnel is needed	2
It is unclear how to compose different models into one	1
Not clear how to mix generated and custom code	1
OCL is badly supported	1
MDA models are hard to maintain	1
It is not clear if MDA can help making development faster	1
It is not clear if MDA can help making development better	1
What about the transition between versions of a system? (can we generate database migration scripts?)	1
It is not clear if MDA can help making development cheaper	1
Is there support for version control on models	0
Code generation cannot be better than code written by smart people	0
Lack of positive experience with MDA	0
How to cope with inconsistency in models (example: contradicting laws in real life)	0
Bad experiences with MDA	0
100% generation is hard	0
Do we model schema’s?	0
Is layout modeled?	0

APPENDIX B TRANSFORMATION PATTERNS

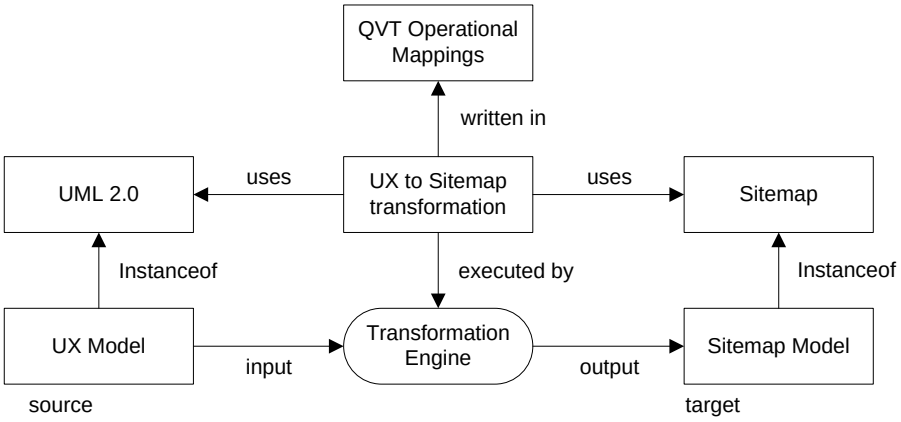
This appendix provides the transformations patterns for all the m2m and m2t transformations we used during this project. Some of these patterns already have been discussed in the thesis. The patterns that are listed here and not discussed in the thesis follow the same pattern as those that were discussed.



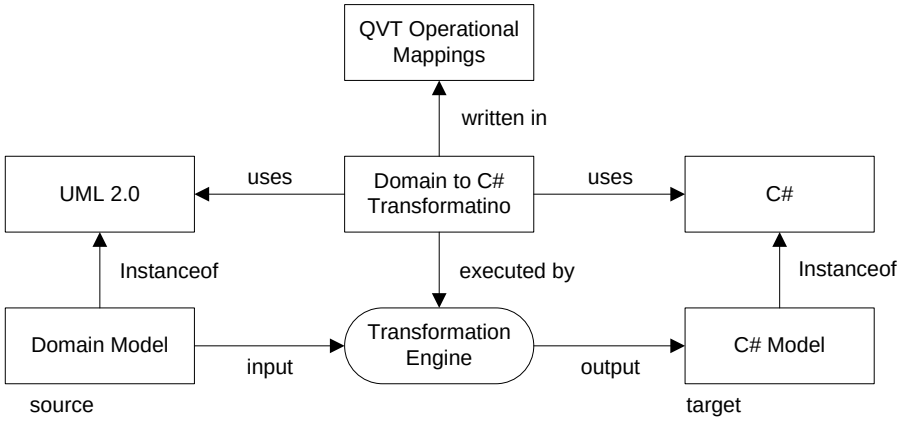
User Experience to ASP transformation pattern



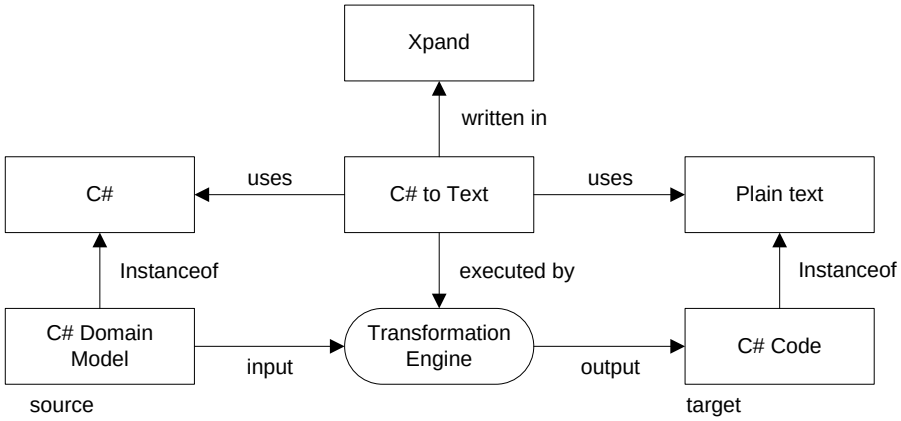
User Experience to C# transformation pattern



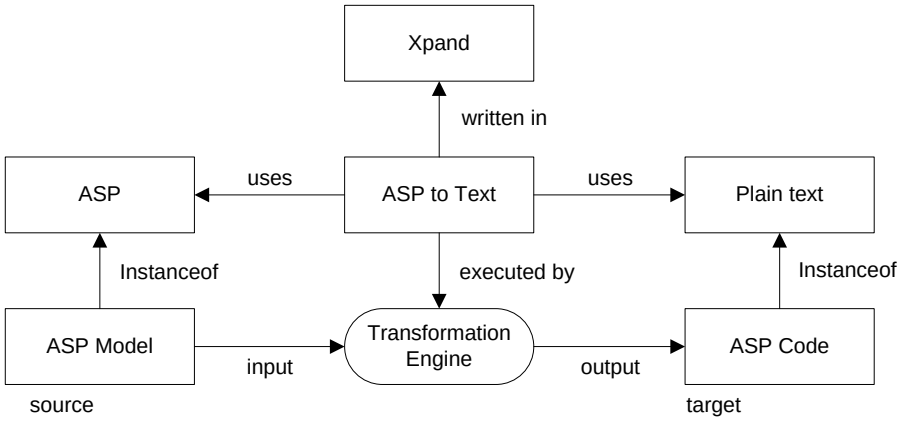
User Experience to Sitemap transformation pattern



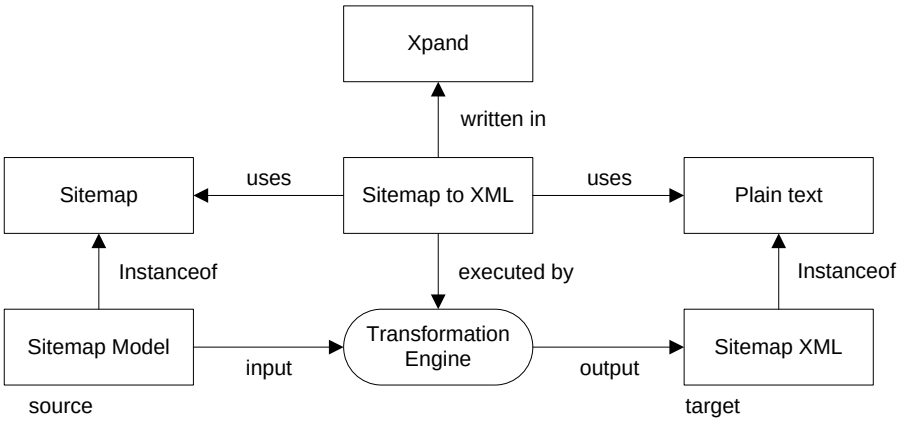
Domain to C# transformation pattern



C# to text transformation pattern



ASP to text transformation pattern



Sitemap to xml transformation pattern

APPENDIX C DOMAIN MODEL

This appendix contains an image of the domain model diagrams we created during this project. In this project one large domain model was created for both use-cases. The model is drawn in Borland Together 2007 and uses the domain model profile we developed during this project.

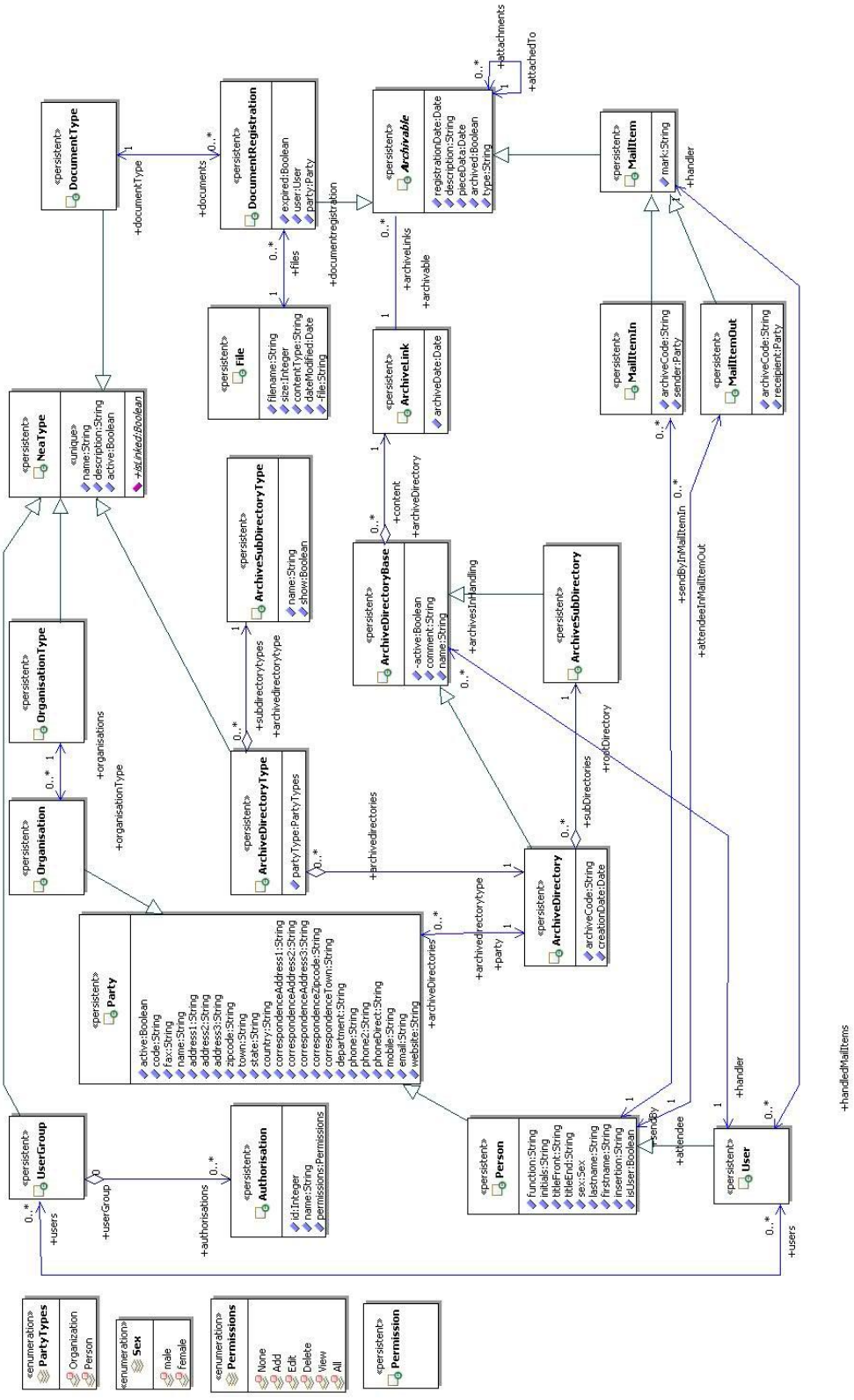


Figure 46: Full domain model

APPENDIX D USER EXPERIENCE UC1

This appendix contains the user experience models we developed during the elaboration phase of this project. The images are created with Borland Together 2007. The models are drawn as UML class diagrams with the user experience profile we developed during this project.

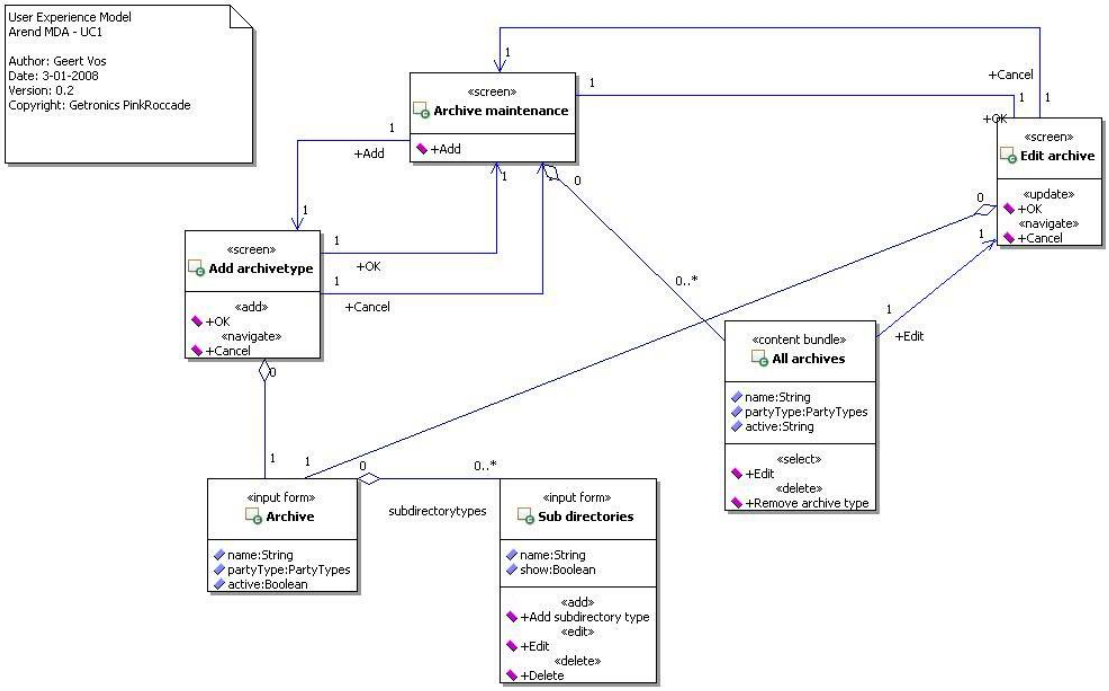


Figure 47: User Experience model for Use-case 1

APPENDIX E USER EXPERIENCE UC2

This appendix contains the user experience models we developed during the construction phase of this project. The images are created with Borland Together 2007. The models are drawn as UML class diagrams with the user experience profile we developed during this project.

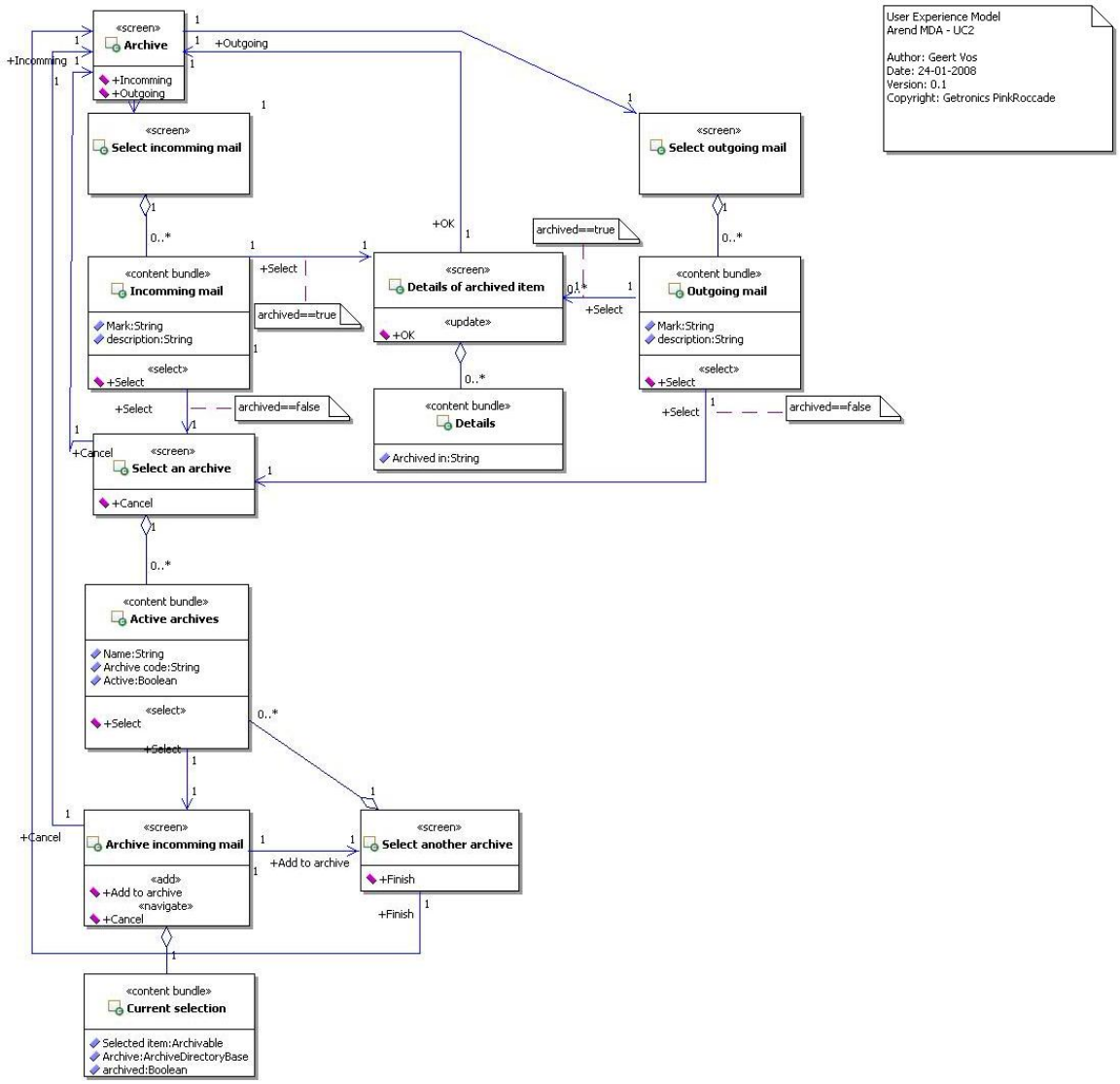


Figure 48: User experience model for Use-case 2, part 1

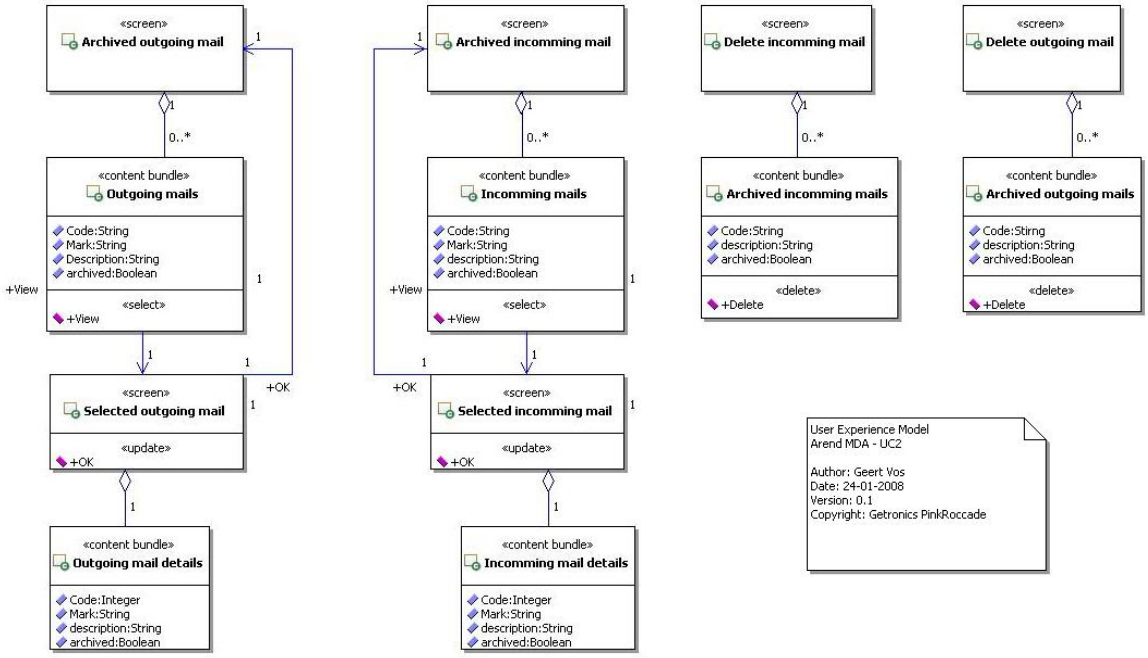


Figure 49: User experience model for Use-case 2, part 2

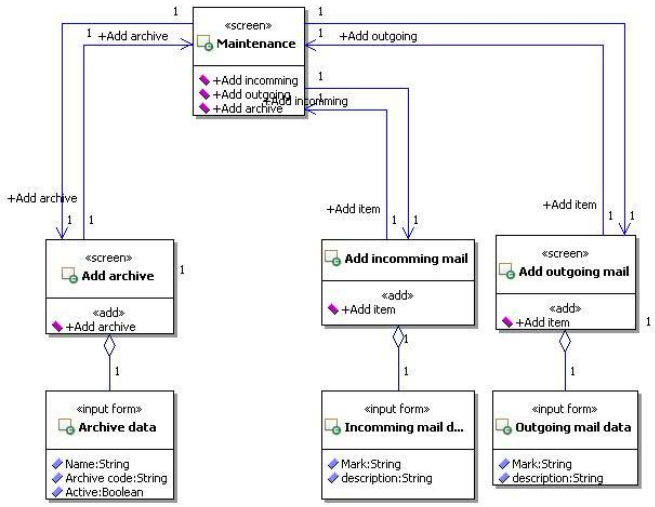


Figure 50: Extra user experience model for testing purposes

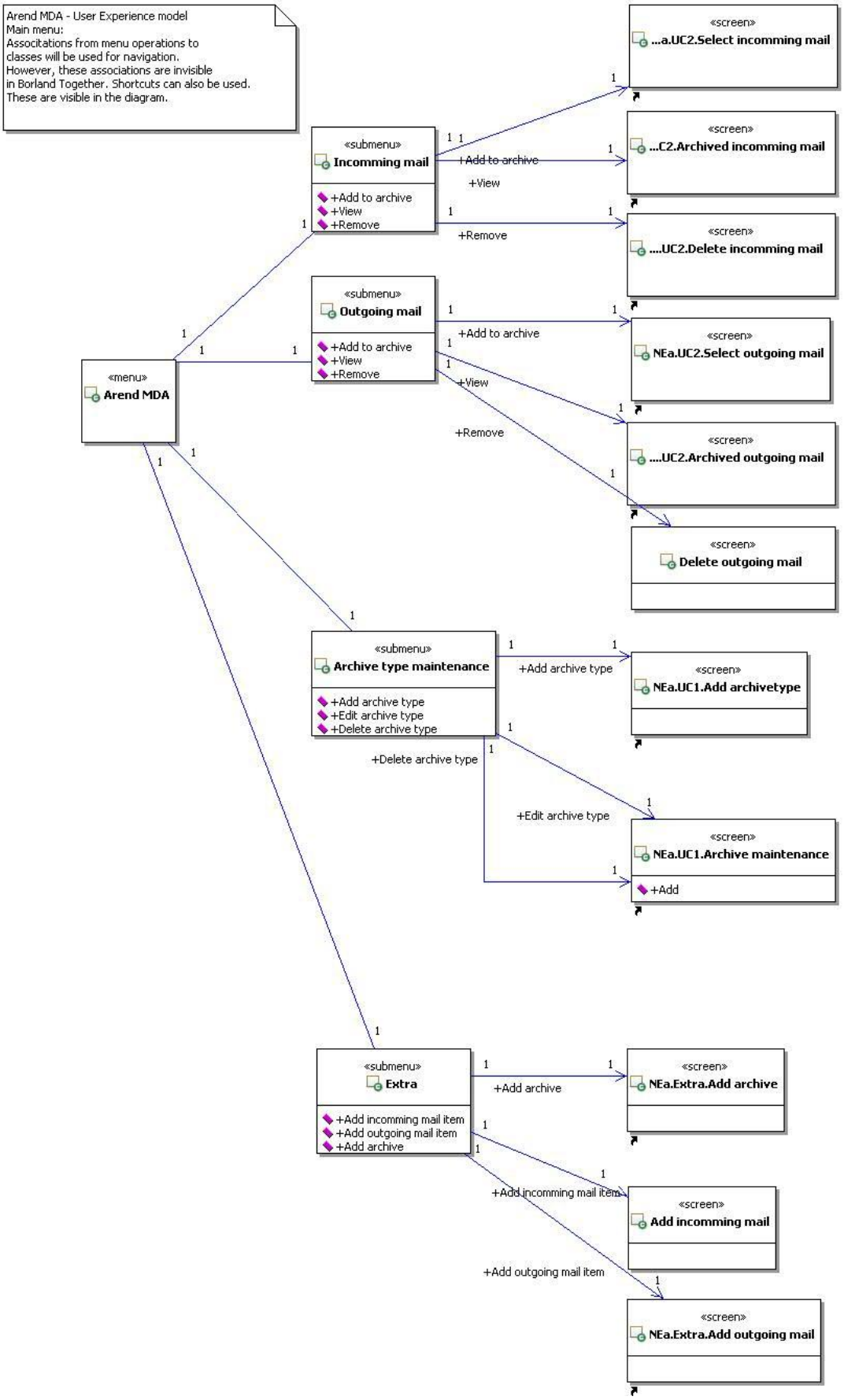


Figure 51: User experience model to model menu structure

APPENDIX F CD-ROM

All documents, plans, illustrations, sources and binaries we used in this project are available on CD-ROM. This CD contains a directory structure that reflects RUP, which we used to structure the overall project.