# Applying Model-Driven Development to Reduce Programming Efforts for Small Application Development

*Research Report*

Launch of the NASA space shuttle Endeavour

Maarten Schilt

# Contents

# List of Figures

# Chapter 1

# Introduction

Due to the raise in complexity of current platforms like J2EE and .NET, software developers need significantly more time and effort to design and implement a software system [56]. Moreover, the increase in complexity and expected quality of large software systems currently developed by software development organizations also have a negative effect on productivity [29]. *Model-driven development* (MDD) aims to reduce complexity in the software development process by raising the level of abstraction at which developers design software systems and thereby increases productivity and the quality of the system under development.

In the history of software development we have already encountered a number of raises in the abstraction level of programming languages. Where we first had to write byte code or assembler to develop executable programs, we now have several languages available—like C# and Java—that allow developers to concentrate more on the actual problem, than on low-level technical problems like memory management, for instance. Hiding these issues simplifies the implementation of a software system and therefore results in an increase in productivity. Model-driven development may be the next step in the search for abstractions in the software development process. In this document we will research the applicability of model-driven development in the software development process of Info Support.

This introduction further contains the following parts. Section 1.1 presents a short definition of the overall project. Section 1.2 lists the research questions that will be answered in this document. Section 1.3 contains information on research related to our project. Section 1.4 describes the intended audience for this document. Section 1.5 thanks the people who contributed to the process of writing this document. Section 1.6 gives the overview of the remainder of this document.

## 1.1 Project Definition

This document is only one deliverable in the project we are doing at Info Support. To give a clear view on the context in which this document is written, we will shortly introduce the overall project in this section. The central research question of the project is defined as follows.

> *How can model-driven development be applied to Endeavour to reduce programming efforts for relatively small and simple applications and keep the option open to expand the application to a full-blown enterprise SOA application?*

At the end of the project we hope to be able to answer the stated research question. Since it is hard to answer the whole question at once, we have divided the project in four separate phases in which we will answer parts of the central research question. We have identified the following phases.

1. *Research phase.* The project is initiated with a research phase to get proper knowledge in the field of MDD. In this phase we will read several papers about the theory, approaches, and applications of MDD. This document is the deliverable of the research phase.

2. *Analysis phase.* In the analysis phase we will analyze applications developed with Endeavour and interview developers who implemented these applications. In this way we hope to identify the parts of Endeavour we can optimize with MDD.

3. *Design and Implementation phase.* The actual implementation of the solution is done in the design and implementation phase.

4. *Verification phase.* In the verification phase we will validate whether our solution has improved productivity compared to the original approach in software development, and how our solution compares with the solution based on frameworks. The project that investigates the framework solution is executed by Bastiaan Pierhagen, a colleague at Info Support.

## 1.2 Research Questions

As we stated in the previous section, this document is the deliverable of the research phase in the overall project. To get the proper knowledge to start the analysis phase, we identified the following research questions that we will answer in this document.

1. *What is the domain in which the problem resides?* To find a plausible solution for the given problem, we should have a clear understanding of the problem domain. We should know what service-oriented architecture is, what small and simple applications are, and how Info Support applies Endeavour to develop business applications with a service-oriented architecture.

2. *What is model-driven development?* We search our solution in the field of model-driven development, and therefore we should explore this field and look for interesting approaches and applications. Only with proper knowledge of the possibilities—and evenly important the limitations—of model-driven development, we can decide whether a certain approach is worth investigating, or not.

3. *What involves applying model-driven development in the problem domain?* We have a problem in a certain domain and a general solution to reduce programming efforts, but how do we map the solution—model-driven development—on the problem domain—business applications with a service-oriented architecture? A complete answer to this question can only be given if we have finished the analysis phase, but we can already define the issues involved in this mapping. The identified issues will give us some guidance during the analysis phase.

When we have answered these three questions, we hope to have gained enough knowledge to correctly execute the remaining phases.

## 1.3 Related Work

While we research model-driven development to reduce programming efforts in developing small Endeavour applications, a colleague and friend of mine, Bastiaan Pierhagen, researches the applicability of frameworks to achieve the same goal. To see which of the two solutions results in the highest increase of productivity, we have included a validation phase in which we will organize an empirical study to compare our solutions. More details on the validation phase will be presented later in this document.

## 1.4 Audience

It should be possible for every person with basic knowledge in the field of software engineering to read and understand this research report, as required by the Delft University of Technology. Before we delivered the final version of this document, it has been reviewed several times to satisfy this requirement.

## 1.5 Acknowledgments

In this section I would like to thank anyone that contributed to the process that resulted in this document. I want to thank Andy Zaidman, my supervisor at the Delft University of Technology, who guided me in writing this report and provided me with useful feedback. I also want to thank my supervisors at Info Support, Dennis Joosten and Marco Pil, for providing knowledge on several topics and for sharing their ideas in this project. Finally I would like to thank Bastiaan Pierhagen, a colleague and friend of mine, for all discussions, ideas, reviews of this document, and motivations.

## 1.6 Overview

The remainder of this document is organized as follows. Chapter 2 thoroughly explores the problem domain and thereby answers the first research question presented in Section 1.2. Chapter 3 introduces the general concepts of model-driven development. Chapter 4 describes the two main approaches in model-driven development, namely Microsoft's

*Software Factories* and the OMG's *Model-Driven Architecture*. Together, Chapter 3 and 4 answer the second research question. Chapter 5 presents the issues involved in mapping model-driven development on the problem domain, and answers the last research question. Chapter 6 defines the project assignment for the remaining phases of the project.

# Chapter 2

# Exploring the Problem Domain

In this chapter we want to explore the problem we have to solve in our project assignment. To find a useful solution for the given problem, we should have enough knowledge of the domain in which the problem resides. Therefore we want to look closer into a few important concepts mentioned in the research question of the overall project. Our research question is presented below.

> *How can model-driven development be applied to Endeavour to reduce programming efforts for relatively small and simple applications and keep the option open to expand the application to a full-blown enterprise SOA application?*

We will look closer into three important aspects of the research question that are related to the problem domain. In Section 2.1 we will define what small and simple applications are and how we measure application size. Since we are dealing with applications with a *service-oriented architecture* (SOA), Section 2.2 will introduce this concept to give a clear view on the structure of the applications in our problem domain. Section 2.3 will describe Endeavour, Info Support's software factory for development of business applications with a SOA. Section 2.4 will present how small and simple applications developed with Endeavour look like. Finally, Section 2.5 contains a summary of this chapter.

## 2.1 Application Size and Complexity

The main research question presented above stated that we will target small applications. To define what a small or large application is, we need a metric to indicate application size. An obvious choice would be to express the size of a software system in the number of *lines of code* (LOC).

Although it might seem logical at first, this approach has quite some disadvantages if we want to determine the time needed to develop a software system [32]. Firstly, we cannot say anything about the complexity of the system, only the length of the source code. Secondly, the size of software now depends on the chosen programming language and the quality of the design of the system. A badly designed system containing a lot of duplicated pieces of source code, probably has more lines of code than a properly designed system with the same

| **Lines of Code** | **Function Points** |
|---|---|
| (+) Objective method | (–) Subjective method |
| (–) View on length of source code | (+) View on length and complexity |
| (–) Language dependent | (+) Language independent |
| (–) Design quality dependent | (+) Design quality independent |
| (–) Only size of code. | (+) Size of code and specifications |

Table 2.1: Comparing LOC and FPs for expressing the size of a software system

functionality. Finally, we cannot take into account the size of the specifications, because we cannot express it in LOC. Given these disadvantages, we do not think that using the number of LOC for size estimation is the way to go.

Another approach is to use *function points* (FPs) as a metric for application size. The *function point analysis* (FPA)—the analysis of a software system to measure its size in function points—is developed by IBM and introduced in 1979 [1]. The number of function points of a software system is based on the functionality offered by the system to its users; these users can either be humans or other software systems that use the system. Each function offered to the user is assigned a number of function points based on the complexity of the function. Three levels of complexity—low, average, and high—are defined to assign these function points to a function. To calculate the total size of an application expressed in function points, we have to sum up the function points of each function offered to the user.

Although FPA has a higher accuracy than counting the number of LOC of the system, it is a language independent method, and it is not restricted to code only, it still has a disadvantage that should be taken into account: since we have to estimate the number of function points for a given function, the FPA method is a subjective method and therefore hard to compute and difficult to automate [32]. We have summarized the advantages and disadvantages of measurement with LOC and FPs in Table 2.1.

Despite its disadvantage we decided to choose FPA as our software estimation metric, because Info Support utilizes this method to express the size of the software they build. Info Support also collects data about their projects based on function points—like the amount of time needed to implement a function point—which allows us to do a comparison with our solution at the end of the project. Now we have chosen a metric to indicate application size, we can define a small application as an application that consists of up to 300 function points.

Now we have defined what a small application is, the question of what a simple application is remains. In general, the complexity of a software system is not related to its size; small applications can contain very complex business logic, while a larger system might contain little logic. However, from years of experience in the field of business application development, Info Support states that the small applications they develop contain little business logic in most cases. From now on we will assume that the small applications we are targeting contain little business logic and are therefore simple applications.

Finally, we want to say something about how we see a reduction in programming efforts.

```
<login>
  <user>...</user>
  <password>...</password>
</login>
```

Listing 2.1: A message for a login service in XML

Given the choice for function points to measure software size, our goal is to reduce the time needed to implement one function point of a software system. Development time is reduced if the developer has to write less code to implement the function point than before. Assuming we need the same amount of total code to implement the function point, we want to reduce the amount of *manually* written source code per function point (mLOC/FP).

## 2.2 Service-Oriented Architecture

As the name implies, a *service-oriented architecture* (SOA) is an architecture based on services. But what exactly is a *service*? In this section we will first answer this question and then describe how we build an architecture of these services.

### 2.2.1 Services

Services are highly coherent software components that can perform one or more functions. A service can be divided into two elements: the *service description* and the *actual implementation* of the service. The service description contains information about available functions, input and output messages, expected behavior, quality attributes, and more. The available functions and messages used for communication with the service, form a contract for the service. Clients can invoke the functions of the service if they satisfy this contract—in other words, if they call one of its functions with the right messages. The concrete implementation of the service is the component offering the functionality specified in the service description.

**Example: Login service** An example of a service is a login service for a web application. The service description of this service specifies, for instance, that the service has one method 'Login' which needs a message containing an encrypted user name and password. In XML this message might look something like in Listing 2.1. While this login service might be implemented as a .NET web service, a J2EE application can invoke the functionality of the service if it satisfies the service contract.

According to Papazoglou, services have three important characteristics [51], which we have listed below.

**Technology neutral** The functions of a service should be invokable for applications running on different platforms. Communication between a client and a service should therefore go through standardized technology available on almost all platforms. This

way a Java application can invoke services running on, for instance, the .NET platform.

**Loosely coupled** Services are loosely coupled to each other, which means that a service does not know anything about the clients that use it and it does not depend on the state of other services.

**Location transparency** Clients obtain service locations from a registry containing service descriptions and location information. Services do not maintain a list of locations of other services, these are always obtained from the registry.

A *composite service* is a service assembled from multiple existing services. The services contained in a composite service are described and invoked as if they were a single service. A composite service could be a travel booking service, where several individual services—like flight booking, hotel reservation, and car rental services—are needed to complete the request. Clients can call the travel booking service, without knowledge of the three underlying services.

A popular implementation of a service is the *web service* [9]. The service description of web services is specified in the *Web Service Description Language* (WSDL) [69], which is a language based on XML. Messages used for communication with web services are *Simple Object Access Protocol* (SOAP) messages [28]. *Universal Description, Discovery, and Integration* (UDDI) [42] is the repository used for publication and discovery of web services. It should be noted that the use of services does not imply the web service technology.

### 2.2.2 An Architecture of Services

Now we have discussed services, we can focus on the SOA concept. A SOA consists of multiple services interacting with each other through some communication protocol. A SOA usually contains three types of elements: the *service provider*, the *service registry*, and the *service client* (or *service consumer*).

The service provider is the person or organization offering a service. The service provider publishes the description of the service in the service registry. Published services in the registry are given a unique identifier, often a Uniform Resource Identifier (URI), to offer the location transparency characteristic mentioned in the previous section. Service clients can search through the service descriptions in the registry and obtain a location to bind to the actual service. Once bound to a service, a client can invoke the functions offered by the service. An overview is given in Figure 2.1.

Loose coupling of services in the SOA allows for independent development of services. When the interfaces and behavior of the services are specified, implementation of the services can be done in parallel by separate development teams. An organization can also choose to outsource the implementation phase to another organization or integrate an existing service instead of implementing it internally.

Although SOAs are nowadays often implemented using web services, it does not imply this technology. A SOA can also be implemented using, for instance, Java Remote Method Invocation (RMI) or .NET Remoting.

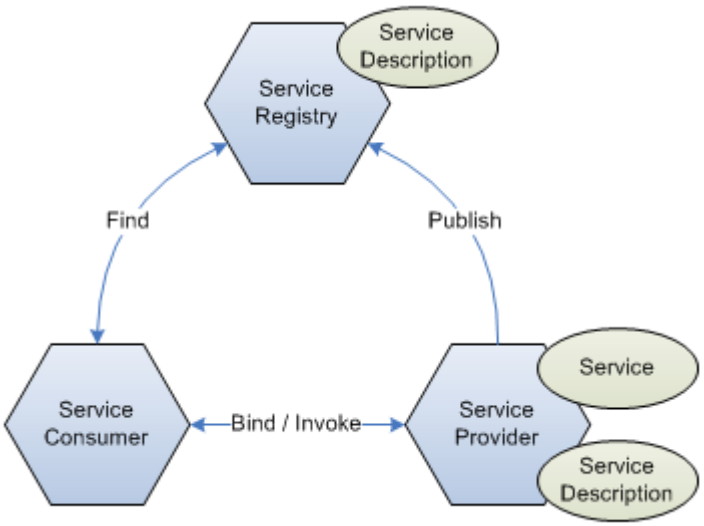



Figure 2.1: Service-Oriented Architecture [19]

## 2.3 Endeavour

Info Support created the *Endeavour* software factory for development of business applications with a service-oriented architecture. Endeavour is more than just a framework for SOA application development; it provides the developer with a project portal, standards, templates, guidelines, best practices, building blocks, architectures, and much more. Both a .NET and a Java version of Endeavour exists. Currently the .NET version of Endeavour is more mature. In this project we will target the .NET variant of Endeavour.

An application can usually be subdivided into components of different types; some related to core business functionality, others to solve a technical problem. Endeavour calls these components *configuration items* [53], which are logical units of functionality and/or data. The following types of configuration items have been identified.

**Business Service** The *business service* contains the business logic for a certain cohered area of data. An example of such an area is the list of customers of an organization; the business service for this area could be a customer management service. Functions of a business service are related to business activities. Therefore a business service does not provide a function called `SaveAddress` to be used if a customer moves from one address to another, but rather something like `MoveCustomer`.

**Process Service** The *process service* implements a business process in the organization. Process services use several business services in their execution. A process service is a composite service as introduced in Section 2.2.

**Integration Service** The *integration service* connects an external system to the internal

communication standard for, for instance, business-to-business (B2B) communication.

**Platform Service** The *platform service* provides functionality needed in the application that is not business functionality. Platform services are often used to solve problems of a technical kind. An example of a platform service is the Exception Management Building Block offered by Endeavour to assist developers in implementing exception handling.

**Frontend** The *frontend* offers functionality to a specified group of end users on the screen. The users of a frontend are always human actors; an external system cannot communicate with the system through a frontend. Frontends can be implemented with technologies as Windows Forms (for .NET rich clients), ASP.NET (for .NET web applications), Swing (for Java rich clients), and Java Server Pages (for Java web applications).

**Service Bus** The *service bus* is the medium through which the services communicate with each other. The service bus provides loose coupling of services and corresponds to the service registry introduced in Section 2.2. No business logic is contained in the service bus. The service bus also ensures that messages sent between services satisfy to a message structure defined in a canonical schema. We want to emphasize that the service bus is not necessarily a physical component in a software system, but can simply be a set of files that specify message structures and service interfaces. We do need some sort of service locater to satisfy the location transparency requirement.

## 2.4 Small and Simple Applications in Endeavour

In the above sections we have defined small applications and introduced the software factory Endeavour, but we did not describe how a small application looks like in Endeavour's SOA environment. In this section we will identify the components usually present in small applications that are developed with Endeavour. We base our statements on the experience Info Support gained from developing small application with Endeavour in the past. For each configuration item identified in Section 2.3 we have presented the number of occurrences in Table 2.2. Given these number of occurrences per configuration item, we can also graphically present how a small SOA application in Endeavour will look like; this is done in Figure 2.2.

## 2.5 Summary

To get a clear view on how we can best apply model-driven development to reduce programming efforts for the development of small and simple applications, we have investigated the problem domain of our project. We have defined a small application to consist of a maximum of 300 function points and stated—based on experience of Info Support—that these application contain little business logic and are therefore simple applications.

| Configuration Item | Occurrences |
|---|---|
| Business Service | 1 |
| Process Service | 0 |
| Integration Service | 0..n |
| Platform Services | 0..n |
| Front End | 1 |
| Service Bus | 0 |

Table 2.2: Occurrences of configuration items in small applications

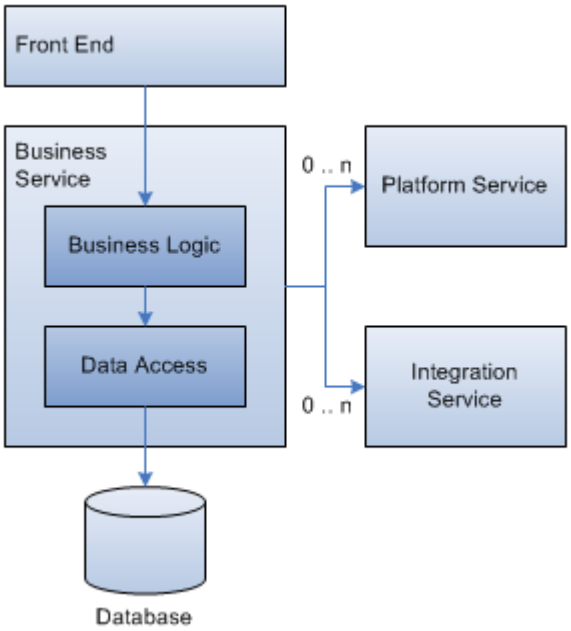



Figure 2.2: A small SOA application in Endeavour

We introduced the concepts service and service-oriented architecture. A service is a coherent software component able to perform several functions that are defined in its interface. An actual implementation of the service contains the functionality for these functions. We stated that services are technology neutral, loosely coupled, and location transparent.

A service-oriented architecture is an architecture of cooperating services that interact through a communication protocol. A SOA can be implemented with, for instance, web services, Java Remote Method Invocation, and .NET Remoting.

The Endeavour software factory supports the developers of Info Support in the development of applications with a service-oriented architecture. We have defined that a small application developed with Endeavour contains one business service, one frontend, a database, and zero or more platform and integration services.

# Chapter 3

# Introduction to Model-Driven Development

In a traditional software development process we usually start with gathering requirements for the system we want to develop. From the list of requirements we specify a number of scenarios and use cases to get a clear view on the behavior of the system. Then we can create an initial conceptual model containing the classes of the system and their interrelations. The next step is to design a more detailed model, which acts as a blueprint for the system under development; this model contains all attributes and methods of classes in the system. Finally, the blueprint model is used as a guideline to write the source code for the system.

*Model-driven development* (MDD) is a development methodology that aims to perform these steps automatically. Ideally, a developer designs a software system at a conceptual level and through a number of transformations the application is generated. If this is possible, it would have several advantages over the original way of software development; the most obvious advantage is of course the increase in productivity, as claimed in [29]. Before we further elaborate on benefits and issues, we start with an introduction to models, meta-models, and model transformations to give a better understanding of MDD, and we identify the effects of MDD on software quality attributes.

This chapter is organized as follows. Section 3.1 presents the basic concepts of MDD: the model, the meta-model, and the model transformation. Section 3.2 deals with the expected effects of MDD on software quality attributes. The benefits and issues of MDD are discussed in Section 3.3. This chapter is summarized in Section 3.4.

## 3.1 Basic concepts

The three central concepts in the field of MDD are the model, the meta-model, and the model transformation. All are needed for a proper MDD approach. In the following three sections we introduce each of these concepts and how they are related to each other.


Figure 3.1: A UML Class Diagram is a model of a software system.

### 3.1.1 Models

In literature, several definitions for the concept of *model* exist [7, 39, 57]. Each of these definitions for model mentions a system and a relation between the system and a model. This relation is called the *RepresentationOf* relation [21]; a model is a representation of a system. A model is created with a particular goal in mind and therefore only information necessary to achieve this goal is included in the model [7]; all other information about the system is considered irrelevant. For example, in an architectural design model of a building it is important to specify the color and shape of each part of the building; modeling the electric wiring through the building does not add to the goal of the model. Models of software systems are often specified in the *Unified Modeling Language* (UML) [46]. A UML class diagram is an example of a model of a software system, as illustrated in Figure 3.1.

Although we are interested in models of software systems in this document, the given definitions do not give this constraint. The system under study can, for instance, be a real-world object like a building or a bridge. Blueprints are an example of a model of a building or bridge. From now on we will purely focus on models of software systems, although parts of the information presented in the remainder of this section might also be applicable on other types of models.

We will now look at characteristics that determine if a model is a good representation of a software systems and present some approaches in classifying models.

**Quality attributes of models** What makes a model a good representation of a software system? Selic identifies five quality attributes a model should possess [58]. We have listed these attributes below.

**Abstraction** Through abstraction a model can hide implementation-specific and too-detailed information. As mentioned in the introduction of this document, software systems get more and more complex; if models hide the complexity of the underlying system, the developers can get a better overview on the system's functionality, which results in higher quality and productivity [29]. The higher level of abstraction also allows for better communication with domain experts, because the model hides implementation-specific information and therefore provides a vocabulary closer to the problem domain [29]. Abstraction is the most important attribute of a model [58].

**Understandability** Just abstracting away non-relevant information is not enough, a model should present its contents in a clear and understandable way. Models can be presented in both textual or graphical form—usually system requirements are presented in structured text documents; an object model of a software system is often in the form of a graph. It is important to choose a presentation approach that results in an understandable model. The choice to present an object model in plain text is an example of a not too clever decision, since this abstraction will not give the developer a clear view on the system.

**Predictiveness** The model should present the system in such a way that it is possible to correctly predict the system's non-obvious characteristics. If we for instance create a model of a bridge to predict the maximum load it can handle without collapsing, we should use a type of model that allows us to obtain this value.

**Accuracy** The important features of the modeled system should be represented accurately in the model.

**Inexpensiveness** A model should be significantly cheaper to construct than the actual system.

**Classification of models** Models can be classified using different approaches. One such approach is to make a distinction between *specification models* and *description models* [39, 57]. The first kind of models is created before implementation of the system and is used to specify how the system under development should look like when it is implemented. The implemented system is only considered valid if it completely matches its specifications defined in the model.

Description models are created to describe existing systems. In this case the model is validated instead of the system under study. The model is valid if all its statements are true for the described system. In software engineering we use specification models for forward engineering, and description models for reverse engineering. In this project we will do forward engineering and therefore we deal with specification models.

Another classification approach is to distinct between *sketchy models*, *blueprint models*, and *executable models* [25, 37]. Sketchy and blueprint models can be utilized for both forward and reverse engineering. Sketchy models for forward engineering are used to communicate ideas with other developers; they present a rough sketch of some part of the system under development. Sketches therefore do not contain all the details of the system. With reverse engineering sketches are used to give an idea on how an existing system works, again without all the details. Sketchy models aim at selective communication instead of complete specification [25].

Blueprint models give the complete specification of a system. In forward engineering the blueprint model specifies all details needed for a programmer to implement the system under development. Blueprint models for reverse engineering completely describe (a part of) the existing system. The main difference between sketchy models and blueprint models is that the former are aimed to be explorative, while the latter are definitive [25].

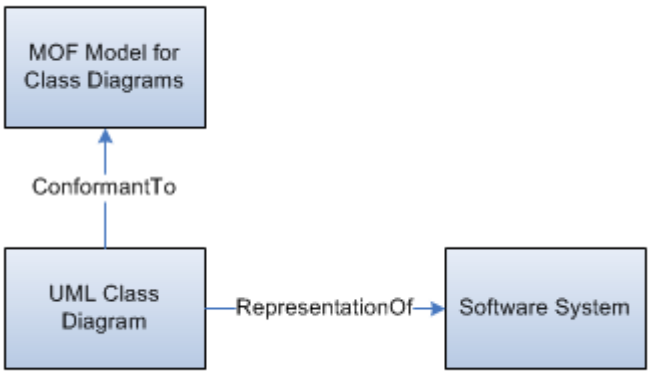



Figure 3.2: The UML Class Diagram conforms to the MOF Model for Class Diagrams.

Executable models are directly compiled to executable code; since the models directly correspond to the underlying code, there is no distinction between forward and reverse engineering. Fowler states that sketchy models are probably used most [25]; the use of executable models is not widely spread [21]. Executable UML is an approach for executable models [36].

### 3.1.2 Meta-models

A specific kind of model is the *meta-model*. A definition of the concept meta-model, taken from [57], is given below.

> *A meta-model is a specification model for a class of systems where each system in the class is itself a valid model expressed in a certain modeling language.*

In short, a meta-model is a model of a *modeling language*. A modeling language is a set of models—for example, all UML models are elements of the UML modeling language [46]. Hence, a meta-model is a model of a set of models [20]. The relation between a valid model and its meta-model is called the *ConformantTo* relation [6, 20]. An illustration of a meta-model and the ConformantTo relation is given in Figure 3.2.

Below we illustrate the concept of meta-model with an example about the UML meta-model for class diagrams. Another example of a meta-model is the C# grammar; all source written in C# should conform to this meta-model to be considered valid.

**Example: A meta-model for UML class diagrams** Figure 3.3 presents an example of an excerpt of the UML meta-model for class diagrams. The meta-model specifies that a class can have zero or more attributes. The class is specified with the `Class` class and contains one attribute to assign a name to the modeled class. The attributes of a class are modeled with the `Attribute` class, which has the attributes `name`, `type`, and `visibility`. We can also create directed associations between classes. These associations have a source and a target

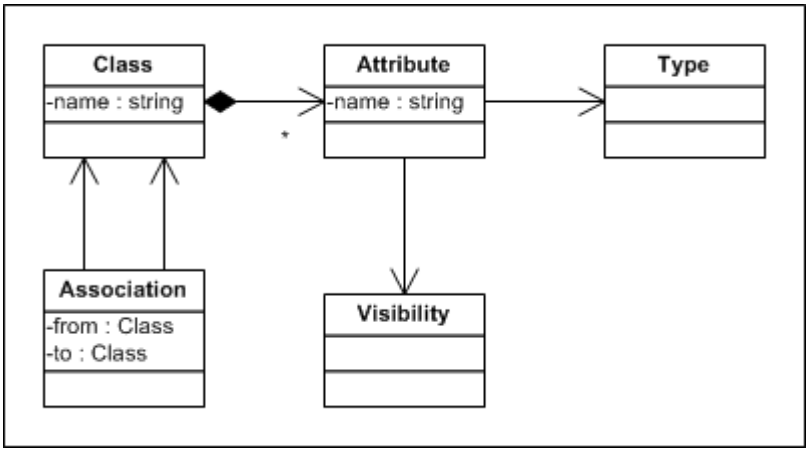



Figure 3.3: An example of a piece of the UML meta-model in UML

class, modeled with the `from` and `to` attributes in the `Association` class, respectively. Note that this example is only an illustration; the real UML meta-model is far more complicated than this simple example.

We have already mentioned that the meta-model is a specific kind of model, therefore a meta-model can also be described in a modeling language. The *Meta-Object Facility* (MOF) proposed by the Object Management Group [43] is an example of a meta-modeling language; it is used, but not limited, to describe the meta-model of UML models.

MDA Distilled gives three reasons for existence of meta-models [37]. Firstly, a meta-model specifies the language you are modeling in, as we have seen in the definitions above. Secondly, meta-models simplify communication about models. If no meta-model exists for a model, a developer has to explain each node, arrow, or other design construct in his model. With a meta-model the developer can, for instance, create a node of a specific node type specified in the meta-model; no extra explanation of the node type is necessary, since it is defined in the meta-model of the language. Finally, meta-models make model transformations possible. We will elaborate on model transformations in Section 3.1.3.

Again, the meta-model is not only a concept in the field of software engineering. An English dictionary is an example of a real world meta-model for the English language. This document is written in the English language and therefore it conforms to the English dictionary. Since an English dictionary is also written in the English language, it conforms to its own specification.

### 3.1.3 Model transformations

With the use of *model transformations*, models can be transformed into other models. Model transformations are the key challenge in MDD, because they facilitate generation of source code from higher-level models. Transformations between models can become very complex [38]. In the introduction of this chapter we have described the traditional

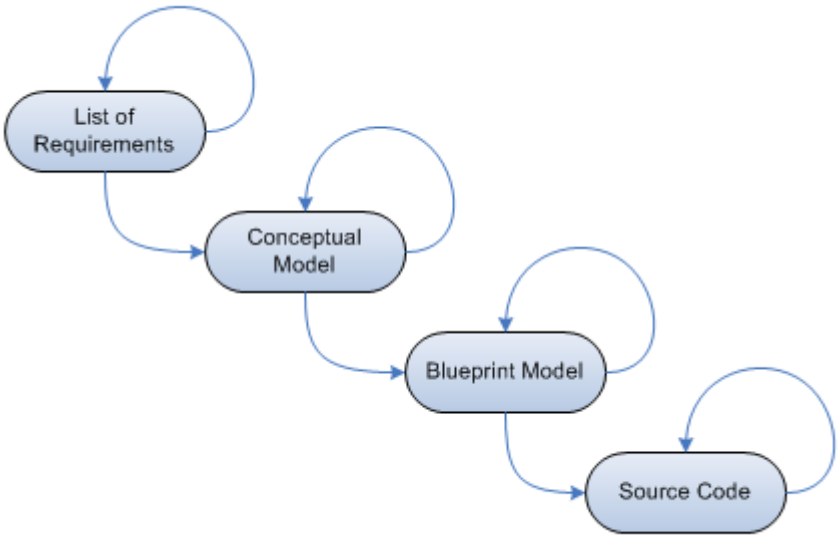



Figure 3.4: Example of models and transformations in the MDD process.

process of software development. If we would use MDD for this process, we would at least need four types of models—a list of requirements, a conceptual model, a blueprint model, and the source code—and three transformations. It is also possible to have transformations that result in a model of the same type as the source model—transforming a blueprint model in a blueprint model, for instance—which we will discuss in more detail later. An overview of models and possible transformations in the MDD process is given in Figure 3.4.

Model transformations take at least one model as input to produce at least one output model. It is also possible to have multiple input or output models [38, 59]. Meta-models of the input and output models are needed to specify transformations between two models; this is illustrated in Figure 3.5. If we want to transform between a UML class diagram and C# source code, we define which element in the source model corresponds to which element in the target model. Given the transformation in Figure 3.5, we would for instance specify that a class node in the class diagram results in a C# class in the source code.

We will now present some characteristics of model transformations, namely how transformations are executed, what kind of models are involved in the transformation, and the directionality of the transformation. We will also describe the desired characteristics of transformation languages and tools.

**Transformation execution** Transformations can either be executed *automatically*, *semi-automatically*, or *manually*. Automatically executed transformations require no input from the user, in contrast with semi-automatic transformations, which ask the developer for parameters to execute the transformation. One approach to give input to transformations is to *mark* models with meta-data [11, 14, 29]. Extra information, unnecessary for the viewpoint

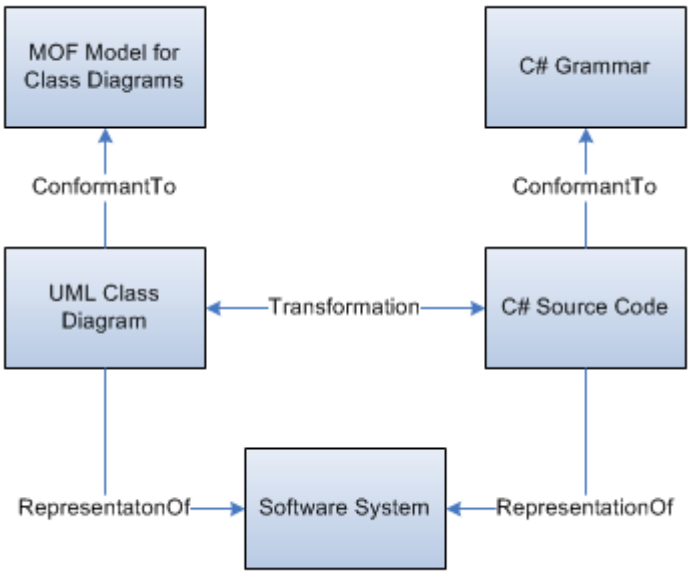



Figure 3.5: A transformation between a UML Class Diagram and C# source code.

```
class Foo
{
        [UniqueId]
        private int id;
}
```

Listing 3.1: An attribute to indicate a unique identifier (C#)

on the system or the level of abstraction, is added to the model to make model transformations possible. An example of marking in UML is to use the stereotype `<<UniqueId>>` on an Integer attribute, to indicate that the attribute represents an identifier which should be unique [39]. Marking in code can be done by adding attributes to programming elements [68]. In Java these attributes are called annotations; C# calls them attributes. An example of attribute usage is given in Listing 3.1. Attributes can also be used to mark business entities, persistent objects, methods that require logging, and much more.

Manual transformations are also possible—think for instance of the transformation from a list of requirements to a conceptual model—but not desirable, because they are more sensitive to faults.

**Transformation type** We can distinguish between *model-to-model*, *model-to-code*, and *refactoring* transformations [11, 14, 38]. A model-to-model transformation converts a model into another model, like a transformation from a UML class diagram to a database

```
public void PrintOrderInfo(Order order)
{
    Console.WriteLine("Order Id: {0}", order.Id);

    // print customer details
    Console.WriteLine("Customer Id: {0}", order.Customer.Id);
    Console.WriteLine("Customer Name: {0}", order.Customer.Name);
}
```

Listing 3.2: A method to print order information (C#)

schema, for instance. These transformations may or may not cross abstraction boundaries; in the former case the transformation is called vertical, in the latter horizontal [38, 59].

Model-to-code transformations generate source code from a given input model. An example is the generation of C# code from a UML class diagram. Note that these transformations are not limited to produce source code in a programming language, it is also possible to generate configuration or deployment files, for instance.

The last type of transformation is the refactoring transformation. Refactoring is defined as a series of small steps, each of which changes the program's internal structure without changing its external behavior [24]. To illustrate refactoring transformations we will now look at an example of the Extract Method refactoring operation on source code. In Listing 3.2 we see a method which prints information of a given `Order` object, namely its identifier and information about the ordering customer. Since a method name should explain the purpose of the method, it would be better to put the code for printing the customer information in a separate method, like in Listing 3.3. This operation can both be executed semi-automatically or manually; automatically performing the Extract Method operation is not possible, since it is hard to guess which code should be extracted and what the name of the new method should be.

In Visual Studio[1] the semi-automatic approach requires you to select the code to be extracted, choosing the Extract Method operation, and specifying the name of the new method. Then Visual Studio generates the code for the new method. Note that refactoring is also applicable on models, like UML class diagrams or state charts [61].

**Directionality** Another important characteristic of a model transformation is *directionality* [38]. Unidirectional transformation are executed in one direction only, in which a source model is converted in a target model. Obtaining a new source model from a modified target model is not possible.

Bidirectional transformations do work in both directions, and therefore have an advantage over their unidirectional counterparts. A bidirectional transformation between UML and C# source code allows to design a system in UML, generate source code from this model, and when the source code is adapted the UML model is updated automatically—this functionality is known as *round-trip engineering* [29]. With a unidirectional transfor-

[1] http://msdn.microsoft.com/vstudio/

```
public void PrintOrderInfo(Order order)
{
    Console.WriteLine("Order Id: {0}", order.Id);

    PrintCustomerInfo(order.Customer);
}

public void PrintCustomerInfo(Customer customer)
{
    Console.WriteLine("Customer Id: {0}", customer.Id);
    Console.WriteLine("Customer Name: {0}", customer.Name);
}
```

Listing 3.3: The refactored version of Listing 3.2 (C#)

mation this last step has to be performed manually, an activity that is fault-sensitive and may be forgotten. Note that bidirectional transformations can be achieved by bidirectional rules in the transformation specification or by defining two complementary unidirectional rules [14]. *Traceability* is a characteristic associated with directionality, which allows the developer to follow a chain of relationships—called a *trace*—between artifacts in different models [29]. Advantages of traceability support are impact analysis—to see how a change in one model affects artifacts in related models—model synchronization, and model-based debugging [14].

**Transformation languages and tools** We will not yet discuss specific transformation languages and tools, but we do give an overview of the characteristics these languages and tools should possess to be applicable. Other characteristics are not required, but are desirable. Sendall and Kozaczynski state that transformation languages must be expressive, unambiguous, and Turing complete [59].

The first two characteristics are probably clear, but what is a Turing complete language? A language is called Turing complete if it has computational power equivalent to the universal Turing machine [66]. Without giving the relatively complex details and a complete description of what a Turing machine is, this means that the language is capable of performing any computational task.

Next to these required characteristics, the authors also list a few desirable characteristics. We combine these characteristics with the list given by Mens, Czarnecki, and Van Gorp [38]; the result is presented below.

**Preconditions** The possibility to describe the conditions under which the transformation produces a meaningful result. This way a tool can indicate which transformations can be applied on a given model, or part of a model.

**Composition** The possiblity to compose a new transformation from existing transformations; this enhances the readability, modularity, and maintainability of the transformation language [38].

**Graphical form** The possibility to design input, output, and the transformation itself using visual means.

**Scalability** It should be possible to design and implement large and complex transformations.

**Conciseness** Meaning that the transformation should contain a minimal amount of syntactic constructs. However, for some often used syntactic constructs we might want to add some syntactic sugar to reduce the amount of work to develop a transformation.

**Validation** The possibility to validate the results of transformations. Although it is an interesting field, we will not focus on validation of transformations in this document; for more information we refer to [22].

## 3.2 Effects on Software Quality Attributes

We have already mentioned that MDD has a positive effect on both productivity and software quality. However, we did not yet specify the exact effects on software quality. As stated in the ISO 9162 standard, software quality is characterized by the following six attributes. The descriptions of the attributes are directly taken from the official document.

**Functionality** A set of a attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.

**Reliability** A set of attributes that bear on the capacity of software to maintain its level of performance under stated conditions for a stated period.

**Usability** A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.

**Efficiency** A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.

**Maintainability** A set of attributes that bear on the effort needed to make specified modifications.

**Portability** A set of attributes that bear on the ability of software to be transferred from one environment to another.

Each of the attributes described above consist of a set of sub attributes. We will focus on the top level attributes, and specify effects on individual sub attributes if necessary.

Because MDD implies designing software at a higher level of abstraction, it has a positive effect on the functionality attribute [29]. The raise in abstraction reduces the complexity of the software constructs with which the developers design a software system, and therefore they can focus more on the actual problem—implementing the desired functionality—instead of dealing with implementation-specific details.

MDD also has a positive effect on the portability of a software system. The system is designed in a model at a higher level of abstraction, from which the source code is generated. The code generator generates code in a certain programming language, but this generator can be replaced with another generator that produces code in another language. If we would, for instance, have a design of a software system in UML, we can use a code generator that converts the UML model to C# code to end up with an application that runs on the .NET platform. In the case we want to offer the possibility to run the same system on a Java Virtual Machine, we can replace the UML-to-C# code generator with a UML-to-Java code generator.

The effect of MDD on the maintainability of a software system depends highly on the type of transformations used and the quality of the code produced by the code generators. MDD can have a positive or negative effect on maintainability, depending on the way transformations and code generation are implemented. A positive effect is achieved when the model transformations used in the MDD process are bidirectional; this way the models of your system will always correctly represent the system [33]. In the case unidirectional transformations are used, the system is generated once from the models, and no changes to the models are made automatically if the source code of the system is altered. If the source code changes, the models have to be modified manually. However, this would also be the case in the original way of software development, so it does not have a negative effect. Note that it is possible to alter the models and then regenerate the application; this involves only changing the models without modification of the source code, which means that the application is maintained at a higher level of abstraction. A negative effect on maintainability is possible though, but this is related to the code produced by code generators. If the generated code is one giant blob of unreadable statements, it does not really contribute to the maintainability of the system. Code generators should generate tidy code, preferably separated from manually written code.

The effect on the efficiency attribute depends highly on the code generated by the code generators. For most of the applications efficiency will not be an issue, as code generators are able to generate code with both performance and memory utilization factors close to manually implemented systems [58]. In the future, these code generators will only get better, achieving even better results.

Finally we have the reliability attribute. In [55] the authors state that Model-Driven Architecture [39] is suitable for achieving a high reliability for the developed system.

## 3.3 Benefits and issues

Before one will decide whether to apply a new software engineering approach or not, it is important to know the resulting benefits and issues. We will present and discuss the known benefits and issues of using MDD in this section.

With MDD we try to generate large parts of the application from models created at a higher level of abstraction. What advantages does this give us over manually programming the full application? An obvious answer is the decrease of development time for the application, since most of the code is generated instead of implemented manually. Although this

is true in most occasions, one should consider that there might be cases where it takes more time to specify and transform the model into source code than to manually writing the code. This can have a cause in the choice of presentation approach for models or the presence of complex transformation that are executed either semi-automatic or manually. Next to the increased productivity, code generation, if properly implemented, also enhances quality and consistency of the source code [11, 29].

Other advantages are a result of the higher level of abstraction at which the developers design the application. The abstraction hides implementation details from developers, which leads to a reduction of complexity of the software artifacts that developers use to design the application [29]. Since current platforms and frameworks are getting more and more complex [56], this reduction is most welcome. The raise of abstraction level also results in a vocabulary closer to the problem domain [29, 57]. This vocabulary allows the developers to communicate about the essential concepts of the system instead of implementation-specific details.

The advantages of MDD sound promising, but to what extent is it possible to generate applications from high-level models? We have already mentioned that transformation rules can become very complex. Some transformations seem impossible to be executed automatically. Think, for instance, of the transformation from a list of requirements of a software system to its initial conceptual model. Automation of this transformation requires a transformation tool that interprets the English language and generates a model in, for instance, UML. Hailpern and Tarr question whether MDD reduces complexity or just moves complexity from development of models to the model transformations [29]. Another issue related to this, is the lack of a standardized transformation language. Without standardization of such languages, there will be no proper tooling support to develop model transformations [29]. To summarize this, model transformations are the heart and soul of MDD [59] and at the same time very hard to develop.

With MDD the complexity of software artifacts with which developers design software system decreases, but at the same time MDD requires more expertise from the developers [29]. MDD relates artifacts in one model with artifacts in other models, and therefore changes in a model have an impact on other models. Developers should be aware of this when changing models.

Finally, there is the initial effort to setup an MDD development process. In the case we use existing technology, we have to select tools and customize these to fit in the process. More effort is needed if we have to develop the meta-models and model transformations ourselves. This will cost even more time and good domain knowledge is required.

In Table 3.1 we have summarized the benefits and issues related to MDD.

## 3.4 Summary

In this chapter we have discussed the basic concepts of model-driven development; the model, the meta-model, and the model transformation. The model is a representation of the software system and specifies only relevant information to achieve its goal, all other information is discarded. This is called abstraction and is the most important quality attribute of

| **Benefits** | **Issues** |
|---|---|
| Increased productivity | Initial effort to setup MDD process |
| Increased quality of code | Lack of standardization and tools |
| Reduced complexity of software artifacts | More expertise required |
| Vocabulary closer to the problem domain | |

Table 3.1: Benefits and issues of model-driven development

good models. Other quality attributes are understandability, predictiveness, accuracy, and inexpensiveness. Models can be classified as specification models or description models. The former is used for forward engineering, the latter for reverse engineering. Another classification approach is to distinguish between sketchy, blueprint, and executable models.

Models should conform to a meta-model to simplify communication about models and allow for model transformation. A meta-model specifies what artifacts—nodes, arrows, etcetera—can be used in a specific kind of model. A meta-model is a model of a modeling language and therefore also described in a modeling language. An example of a modeling language for meta-models is MOF.

Model transformations transform a source model in a target model. To specify model transformations we need meta-models for both the source and target model. Transformations can be executed automatically, semi-automatically, or manually, and may or may not cross abstraction boundaries. We can also distinguish between model-to-model, model-to-code, and refactoring transformations. Important is to consider directionality, a characteristic that influences the maintainability of the solution. Unidirectional transformations can only generate a target model from a source model, the other way around is not possible. Bidirectional transformations work in both directions, which has a positive effect on maintainability, since the source and target models will always be up-to-date.

We have also presented the effects of MDD on software quality attributes and discovered that MDD can have a positive effect on functionality, portability, and maintainability. MDD can have a negative effect on maintainability if code generation is not implemented properly. The negative effects on efficiency and reliability are negligible.

Finally we listed the benefits and issues of using MDD for software development. An overview is given in Table 3.1.

# Chapter 4

# Approaches in Model-Driven Development

In the previous chapter we introduced the essential concepts in model-driven development. But how is this theory applied in practice? This chapter will present the two main approaches in MDD, namely Microsoft's *Software Factories* (SF) and the Object Management Group's *Model-Driven Architecture* (MDA). In both approaches software is designed in models at a higher level of abstraction from which code is generated to produce the actual system; this process should increase both productivity and code quality. However, both approaches use different methods to specify these models. Where MDA utilizes UML to create models in, SF build on domain-specific languages (DSLs). We will elaborate on both approaches and their ways of software design.

The goal of this chapter is not to give an extensive comparison of the two methodologies, but rather to present an introduction to both. We do present the main differences between MDA and SF, as identified in [15].

Beside Model-Driven Architecture and Software Factories, there are other, less popular approaches in MDD, namely Domain-Oriented Programming [63] and Agile Model-Driven Development [2].

This chapter is organized as follows. Section 4.1 discusses Microsoft's Software Factories. Section 4.2 introduces Model-Driven Architecture from the Object Management Group. Section 4.3 gives a comparison of both approaches, based on characteristics. Section 4.4 contains a summary of this chapter.

## 4.1 Software Factories

Microsoft proposes *Software Factories* (SF) for rapid development of a specific type of application. With this methodology, Microsoft hopes to increase capacity by moving from craftsmanship to manufacturing in the software industry. With the focus on economies of scope, where multiple unique but similar products are developed, they build on software product line concepts to industrialize the software industry. A software product line supports development of product families, in a sense that it separates commonality and variability.

Large parts of members in a product family are usually identical and therefore reusable. Reusable assets can be requirements, architectural design, planning, code, testing, and much more [3]. As we will see in this chapter, a software factory is more than just a software product line.

Greenfield and Short provide a definition of a software factory [26], which is presented below.

> *A software factory is a software product line that configures extensible tools, processes, and content using a software factory template based in a software factory schema to automate the development and maintenance of variants of an archetypical product by adapting, assembling, and configuring framework-based components.*

This extensive definition probably raises more questions than that it answers any. What exactly is a software product line? What is a software factory template or schema? In this chapter we will answer these questions and explain how MDD is applied to realize software factories.

The remainder of this section is organized as follows. Section 4.1.1 describes the dimensions of innovation on which software factories build. Section 4.1.2 elaborates on domain-specific languages, the languages in which models of a software system are defined for a specific domain. Section 4.1.3 gives a detailed explanation of how software factories work. Section 4.1.4 introduces tools available for development of software factories.

### 4.1.1 Dimensions of Innovation

Software Factories build on three main dimensions of innovation—abstraction, granularity, and specificity—as presented in [26]. To innovate along these dimensions, SF makes use of two development methodologies, *model-driven development* (MDD) and *component-based development* (CBD). We will now discuss each of the dimensions of innovation.

*Abstraction* is an important aspect of models in MDD, as we have seen in Section 3.1.1. The higher the level of abstraction, the more details of the system are omitted in the models, which reduces complexity but narrows the scope of their application. MDD is applied in SF to model at a higher level of abstraction and to generate application source code. SF does not utilize UML for modeling, because it does not provide the required capabilities for domain-specific modeling [26]. Instead, SF uses domain-specific languages to raise the level of abstraction. We will look into domain-specific languages in Section 4.1.2.

*Granularity* is a measure of size of software constructs used in the abstraction. The higher the granularity, the bigger the improvement on reusability, because coarser grained components encapsulate more functionality than finer grained components, and have fewer dependencies. Take for example a software company that is specialized in developing software systems for organizations that offer products to customers. Each of the software systems developed for these organizations will probably contain functionality to store and manage customer information—like a customer name, billing address, telephone number, email address, and probably much more. If the granularity of components in one such software system would be low, the code that implements the functionality for customer management

would probably be scattered across numerous source files of the application. This will not allow for reuse of this code in another application, since rewriting the code will probably be faster than finding all the code related to customer management in the existing application. However, if we would gather all code related to customer management in one coarse grained component—a service, for instance—we could achieve reusability of this functionality. In a new project, the developer only has to include the customer management component to make use of the customer management functionality.

Ideally, a software developer integrates several existing coarse grained components into one new software system, which would drastically shorten the development time of applications. However, since each application probably has some aspects that are not captured in existing components, the developer will still have to write application-specific code. It is also likely that the developer has to write code to glue the several components together. High granularity is achieved with CBD, a development methodology that aims at independent development of coarse-grained components to improve reusability. A more thorough examination of CBD falls out of the scope of this document.

The last dimension is *specificity*, which ranges from general-purpose to domain-specific. For software factories this is perhaps the most important dimension of innovation, according to Greenfield and Short [26]. Specificity defines the scope of an abstraction; the narrower this scope, the higher the possibility for reuse. However, abstractions with a higher specificity are usable in fewer products. UML is an example of a modeling language that raises the level of abstraction with a broad scope; it is possible to design an object-oriented piece of software in any domain available. A DSL however targets a specific domain—like business applications, for example—and is useless in other domains. Software factories inherit a high specificity from software product lines, on which they are based. Software product lines are described in Section 4.1.3.

### 4.1.2 Domain-Specific Languages

A *domain-specific language* (DSL) is a programming language that specifically addresses a particular problem domain. A more extensive definition is given by van Deursen, Klint, and Visser [16].

> *A domain-specific language (DSL) 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 domain.*

An example of a DSL is the Structured Query Language (SQL) often applied to fetch, store, update, and delete data in a relational database. SQL is a DSL because it targets a specific problem domain—managing data in a database—and is restricted only to this domain. Outside its domain, SQL is useless; it is impossible to use SQL for a domain like user interface development, for instance. SQL is an example of a DSL with a textual syntax, but graphical DSLs also exist.

An example of a graphical DSL is the form builder in Microsoft's Visual Studio for development of graphical user interfaces for .NET applications [26]. The drag and drop

functionality allows you to quickly position controls in a form. Other properties of controls are displayed in a list and can be changed if needed. Behind the scenes, Visual Studio generates the necessary code to create the form at runtime. It is obvious that this works a lot quicker than positioning controls by setting location properties in code manually and running the application to see the result. Again, the form builder targets only one specific domain—user interface development—but is more powerful in this domain than a general-purpose programming language. It is obvious that a form builder is useless in other domains.

**External and Internal DSLs** Fowler distinguishes between external and internal DSLs [23]. *External DSLs* are languages written in a different language than the language of the main application, in contrast with *internal DSLs* which are written in the same language as the source code of the main application. Examples of external DSLs are Little Languages in Unix [17] and the XML configuration files that are often used in Java and .NET projects. Internal DSLs can be created in every programming language. Olsen describes how you can create an internal DSL in Ruby [49], a language very suitable for internal DSLs because of its clean syntax and meta-programming possibilities. There are several issues that influence the choice between an internal and external DSL. We will now discuss the consequences of the choice for one of both types of DSLs.

Utilizing an external DSL in a software factory requires you to design its syntax and semantics, and to write an interpreter that parses and executes code written in the DSL. The freedom to design the language is, of course, an advantage, but you do have to have the capabilities to write the interpreter for your new language. Parser generators can assist in the process of creating a parser, but even with assistance this can still become a complex task. The more complex the DSL, the harder it will be to write a parser for the language. In some cases things are easier. If you would, for instance, base your DSL on XML, you can make use of the XML processing functionality available in most general-purpose languages. However, this will limit the freedom of design of your language, since you will have to live with the tag notation used by XML. In both cases you do have to write an interpreter for your DSL, and whether simple or not, this does mean extra work and is thus a disadvantage.

Writing a new language from scratch has more consequences. First, you will have to design the syntax and semantics of the language. Although DSLs are meant to be simple and comprehensive, it is hard to write a DSL which satisfies these properties, without losing its expressive power and specificity [16, 23]. Secondly, there is a lack of tool support for the new language. For writing source code, developers at least expect syntax highlighting—the coloring of language elements, like keywords—in an editor. Nowadays it is common to use an *integrated development environment* (IDE) for software development. Next to syntax highlighting, these applications often support refactoring, semantic editing—like automatically showing a list of available methods of an object when you type the '.' character in the Visual Studio IDE—and debugging. These tools will not be available for a newly designed language and are hard to develop. Finally, the developer has to learn yet another language. However, since DSLs are usually simple [23], this last disadvantage is not a very big issue.

The last issue presented by Fowler is related to the evaluation of the code written in the DSL. Since internal DSLs are written in the same programming language as the main application, they are usually compiled together with the other source files. So evaluation

occurs at compile-time. External DSLs can be evaluated at compile-time or at runtime. In the first case a compiler transforms the DSL code to code in a general-purpose language, which is then compiled to get an executable system. When evaluated at runtime, there is no need to recompile the whole system if a change is made to code written in the DSL. Think of the XML configuration files in, for instance, a .NET project. If you change a configuration parameter in the configuration file, you do not have to recompile the system to apply the changes in the configuration of the system; just running the application again is enough.

It is hard to say that one approach is better than the other. The choice between an internal and external DSL depends highly on the situation at hand. If time is limited you will probably not choose to develop an external DSL, since this requires much more time than implementing an internal DSL. However, if time is not an issue and the skills to develop a parser for an external DSL are available, you might want to design your own DSL.

### 4.1.3 How Software Factories Work

Since software factories are defined to be model-driven product lines [26] we first explain what software product lines are, before we elaborate on the way software factories work. A *software product line* is used to create members of a *product family* of software systems. Systems in such a family share common features, but are not completely identical. Product lines take advantage of the known commonalities and variabilities.

A product line consists of four basic concepts, as presented in Figure 4.1. The product line needs input in the form of a collection of *software assets*. These assets are requirements, components, test cases, and frameworks, for instance. Since we are dealing with product families, the assets should be configurable to allow creation of all products in the family. A *decision model* describes the variabilities between products in the product line. For each variability in the decision model, a decision—called the *product decision*—has to be made. Each product in the product line is uniquely defined by its product decisions. The actual configuration of software assets based on product decisions is called the *production mechanism*. The result of the production mechanism is a specific member of the family of products supported by the product line.

Now we know the essentials of how software product lines work, we can focus on software factories. Software factories are based on three main ideas: the *software factory schema*, the *software factory template*, and an *extensible IDE*. In [27] these concepts are explained by comparing them with a recipe, a bag of groceries, and a kitchen, respectively. A recipe contains several ingredients needed to cook a certain meal; likewise, a software factory schema lists several items—like projects, source code directories, and configuration files—needed to produce a member of a product family. It also describes which DSLs are utilized and how the models designed with these DSLs are transformed into code, other artifacts, or models at a lower level of abstraction. The software factory template contains the items specified in the software factory schema, like a bag of groceries contains the ingredients listed on the recipe. Items in the template are used to build members of a product family; examples of such items are patterns, templates, frameworks, visual DSL editing tools, scripts, and style sheets. Finally, the ingredients are cooked in the kitchen, which results in the meal; this kitchen can be compared with the extensible IDE. The items in the

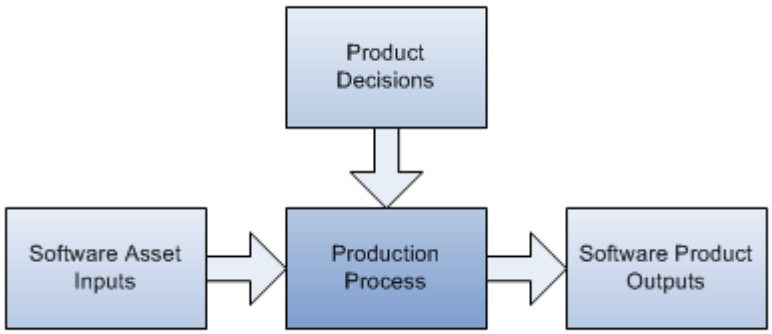



Figure 4.1: Concepts of a software product line [34]

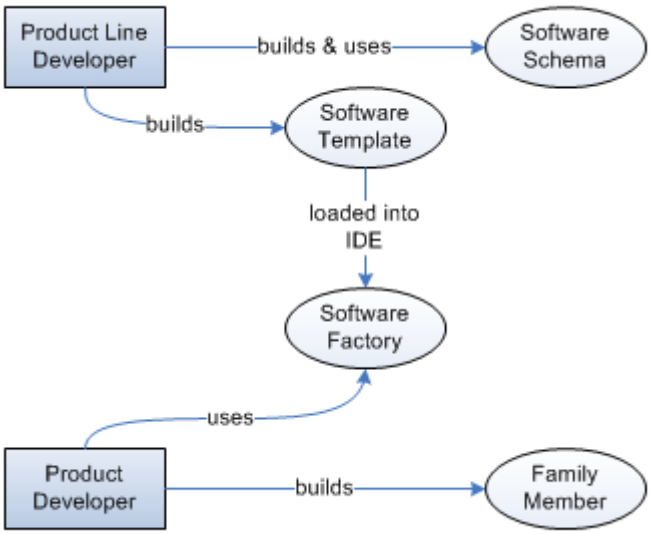



Figure 4.2: Overview of a software factory [27]

software factory template—the bag of groceries—are loaded into the IDE—the kitchen—which becomes a software factory for the product family. An overview of the software factory is given in Figure 4.2.

If we look at the descriptions of the software product line and software factory given above, we see a lot of similarities and some differences. Both the product line and the factory use configurable software assets to create one specific product in a family of products. The difference lies in the way these assets are configured. Software product lines do not specify how configuration of software assets is done. In a software factory, embedded DSLs are utilized to configure the assets defined in the software factory schema. The soft-

ware factory schema corresponds to the decision model for product lines, but the product decisions in a software factory are taken in a model-driven approach. This is why software factories are defined as model-driven product lines [26].

### 4.1.4 Tools for Software Factory Development

Microsoft provides *DSL Tools* [64] to support the development of software factories. DSL Tools allow developers to create custom graphical designers and code generators that are integrated into the Visual Studio IDE. This way Visual Studio can be completely customized to produce one specific member of a product family. The suite of tools include, amongst others, the following elements.

- A graphical DSL designer, which you can use to edit and validate DSL definitions.
- A set of text templates for generating the code of a graphical designer from DSL definitions, where these templates have been designed to produce code that can be further customized by hand.
- A text template engine and framework that makes it easy to write and execute templates that generate source code and other text files from information held in models.

In the case a DSL is not necessary, it is also possible to use the *Guidance Automation Toolkit* (GAT) to develop a software factory [54]. GAT allows the developer to extend the Visual Studio IDE with *recipes*, *wizards*, and *templates*. A recipe is a list of atomic actions performed in a specified sequence—for instance, adding an option to the project context menu that adds a class implementing the Singleton pattern to the project. A wizard gathers information from the developer in a number of successive steps. Each step is displayed as a page in the wizard. The values gathered by a wizard can be used to execute a recipe. Templates allow the developer to define custom Visual Studio solutions, which already contain some predefined projects, for instance. Think of a solution for a business application based on the Model-View-Controller pattern; a solution for this application will probably need a separate project for the model, the view, and the controllers. With a template this structure can already be defined. Templates can also be associated with recipes in a sense that when the template is unfolded, the specified recipe is executed.

We only have a small impression of the possibilities these tools offer. If we decide to choose SF to build our solution on, we will more thoroughly investigate DSL Tools and GAT.

## 4.2 Model-Driven Architecture

The Object Management Group (OMG) proposes *Model-Driven Architecture* (MDA) [39] for raising the level of abstraction at which developers design their software. Instead of DSLs, MDA utilizes the Unified Modeling Language (UML) [46] to create models of software systems. Although the main focus is again on increasing developer productivity, MDA

also aims to improve portability, interoperability, maintenance, and documentation [33]. In this section we will introduce the MDA process and the technologies used in this process.

This section is organized as follows. Section 4.2.1 presents the basic principles of the MDA development process. Section 4.2.2 describes how the concepts model, meta-model, and model transformation, as introduced in Chapter 3, are utilized in MDA. Section 4.2.3 introduces the technologies adopted by the OMG to support the MDA process.

### 4.2.1 Basic Principles

The MDA process does not differ much from the traditional way of software development [33]. Both approaches go through the same phases, but the differences lie in the artifacts used in these phases. Diagrams in the traditional software development process may also be created with UML, but only serve as a sketch or blueprint model, as defined in Section 3.1.1. These diagrams and the source code of the application are all created manually. With MDA the developer only designs a platform independent model, which can be compared with a sketch model. From this model a new model is generated which is specific for a given platform and contains enough information to implement the source code of the system—it is a blueprint model. The blueprint model can now be used to implement the system manually, but MDA proposes to automate this step and generates the source code.

This approach has several advantages over the traditional software development process [33]. Firstly, productivity is increased since a lot of manual work is automated. Secondly, the portability of the software system improves because the software is designed in a platform independent way. If we would like to have a Java version of an existing .NET application developed with the MDA process, we would only have to generate a new model specific for Java from the initial platform independent model. From this new model we can generate the Java code. Of course we do need to write some source code in the end—usually not all logic for a software system can be captured in UML models—but it is a lot faster than starting a new development process from scratch. Finally, we get better maintainability because our system exactly matches its design models. We can change the system by altering a high-level model and regenerating the source code—which leaves the consistency between models and source code intact—and good tools also allow changes to the source code, or a platform-specific model to be propagated to the higher-level models. Another improvement in maintainability is achieved if we automatically generate documentation for the source code.

### 4.2.2 Models, Meta-Models and Transformations with MDA

We will now present the different types of models used in a MDA process. MDA distinguishes between *Computation Independent Models* (CIMs), *Platform Independent Models* (PIMs), and *Platform Specific Models* (PSMs) [39]. A description of each of these model types is given below.

**Computation Independent Model** Models of this type are, as their name implies, independent of any computation and do not show any structure of the software system it describes. CIMs are sometimes called *domain models*, as they are used to model a

problem in a specific domain. The main purpose of CIMs is to bridge the gap between domain experts and software developers [39].

**Platform Independent Model** A PIM describes the system in a platform independent way. Models in UML are, for instance, independent of the programming language in which the system will be implemented, as long as it is an object-oriented language.

**Platform Specific Model** The PSM is a representation of the same system as specified in the PIM. The PSM is obtained from the PIM through transformations. A PSM can be an implementation if it specifies enough information for the system to be implemented.

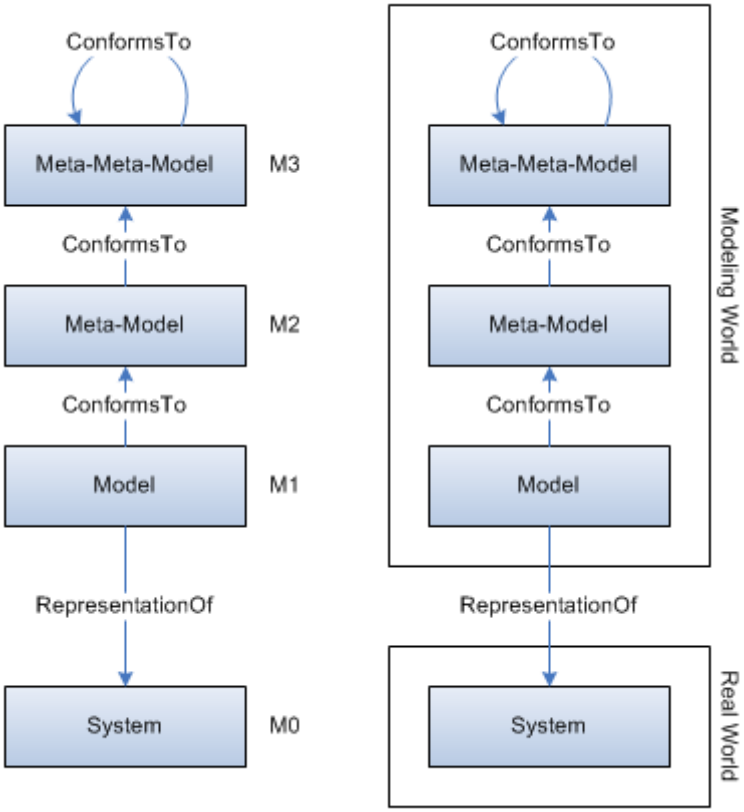
From the above models, automatic transformations are only possible between PIM and PSM. A transformation from a CIM to a PIM is not possible, since this requires a choice of which parts of the CIM should be supported by the software system; this choice is always done with human intervention [33]. A transformation from PSM to source code is also possible, as the PSM is actually a blueprint model and therefore contains all information needed to implement the designed software system. Since the transformation from CIM to PIM cannot be executed automatically, as stated above, the automatic MDA process starts with a PIM. This PIM can then be transformed in several other PIMs, before it is transformed to a PSM. The resulting PSM can then be transformed to new PSMs, and finally into source code. The PIM-to-PIM and PSM-to-PSM transformations are refactoring transformations, as presented in Section 3.1.3.

To facilitate the transformation between different types of models and between the PSM and source code, the MDA needs meta-information of the used models. Therefore MDA comes with an architecture that has several levels of meta-models. The MDA architecture consists of four layers, as illustrated in Figure 4.3(a). The bottom layer $M_0$ is the actual system we are modeling. This system is represented by a model, which is defined at layer $M_1$. The modeling language used to model the system is defined in a meta-model at layer $M_2$; the model in $M_1$ conforms to this meta-model. Finally we have the meta-meta-model—a model that defines how the meta-model in layer $M_2$ should look like—that resides at layer $M_3$. The meta-meta-model conforms to itself.

Bézevin proposes to rename the four layer MDA architecture to the 3+1 layer architecture [6]; Figure 4.3(b) illustrates how this architecture would look like. The new architecture separates between the real world and the modeling world. Models can specify the system from several viewpoints, implicating multiple models for one system. By dividing the architecture in two separate parts—the modeling world and the real world—we obtain a more general organization; Figure 4.4 illustrates this by adding another 'modeling world', the programming language C#.

### 4.2.3 Enabling Technologies

The OMG adopted a number of technologies to support the MDA process. These technologies are the Unified Modeling Language, UML profiles, the Meta Object Facility, and the transformation language QVT. We will now give a short introduction to each of these technologies.




(a) The original (b) The 3+1 alternative

Figure 4.3: The MDA Architecture

**The Unified Modeling Language** The *Unified Modeling Language* (UML) is a specification language used to, but not limited to, create models of software systems. UML contains several types of diagrams to create different views on an application. The diagrams are categorized in three main categories: structure diagrams, behavior diagrams, and interaction diagrams. A structure diagram specifies what things—classes, objects, packages, components, etcetera—should be in the system being modeled. Class diagrams, component diagrams, and object diagrams are examples of structure diagrams. Behavior diagrams emphasize what should happen in the system being modeled. Examples of behavior diagrams are activity diagrams and use case diagrams. The last category, interaction diagrams, is a subset of behavior diagrams and describe the flow of control and data of the things in the system. This category contains, among others, the communication diagram and sequence diagram. For more detailed information about the different diagrams available we refer to [25].

UML models are serializable to XMI [48], another OMG standard. XMI is an abbre-

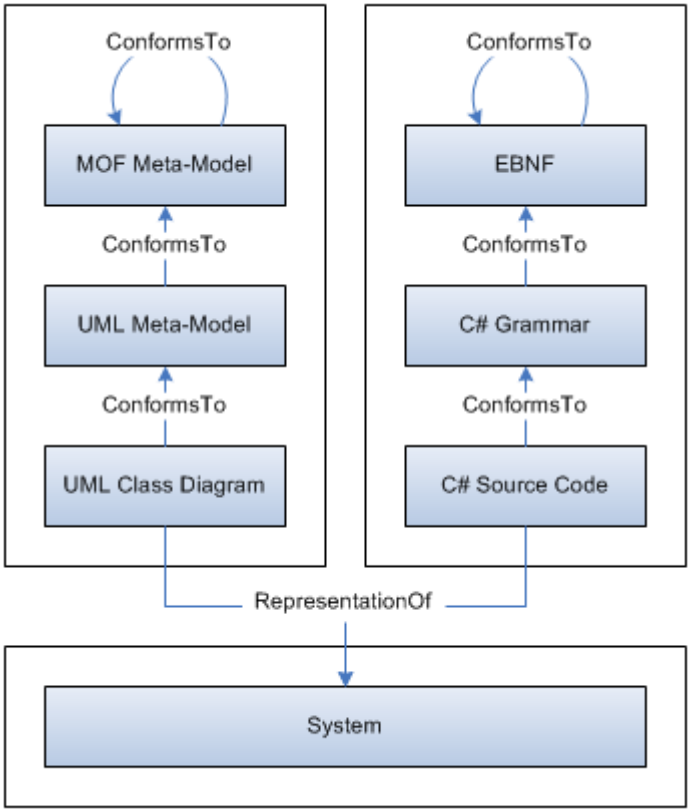



Figure 4.4: Multiple modeling worlds corresponding to a software system

viation for *XML Metadata Interchange* and is, like the name implies, based on XML [67]. These serialized files can be used as an input for transformation tools or as input for another UML tool.

**UML Profiles** *UML Profiles* allow extension of the UML meta-model with user-defined constructs. This provides developers with the possibility to tailor the UML to a specific domain. An example of a UML Profile is a profile for web services, which allows the developer to design web services, service contracts, ports, etcetera [60]. A UML Profile is a specification that does one or more of the following [47]:

- Identifies a subset of the UML meta-model.
- Specifies well-formedness rules beyond those specified by the identified subset of the UML meta-model.
- Specifies standard elements beyond those specified by the identified subset of the UML meta-model.

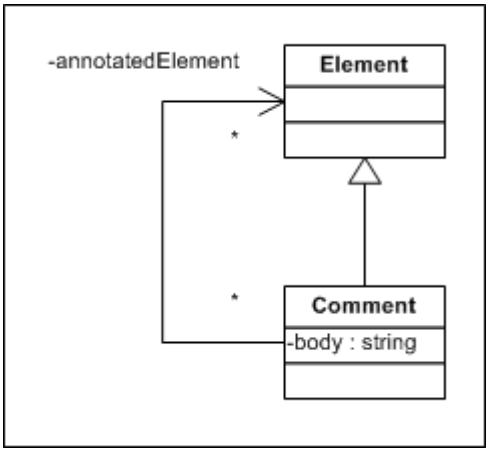



Figure 4.5: A piece of the UML meta-model defining comments.

- Specifies semantics, expressed in natural language, beyond those specified by the identified subset of the UML meta-model.
- Specifies common model elements, expressed in terms of the profile.

As we can see from the list above, it is both possible to define a subset of the UML meta-model or to extend the meta-model with new elements, rules, and semantics. The former involves limiting the developer of using certain types of diagrams, elements, or connectors. The latter allows defining new elements—like web services, service contracts, and ports, as mentioned above—to be used by developers in the design process.

**The Meta-Object Facility** The *Meta-Object Facility* (MOF) is a language that contains a set of modeling constructs that a modeler can use to define and manipulate a set of interoperable meta-models [37]. In other words, MOF allows specification of meta-models. The meta-model of the UML is, among others, defined in the MOF. The concepts available in MOF to design meta-models are types, generalization, attributes, associations, and operations.

Figure 4.5 shows an excerpt of the UML meta-model which defines the class `Comment`. The `Comment` class represents the comments attached to model elements in a UML model, and therefore has a body—which holds the comment—and is associated to an element—which is described in the comment. Since an element can have multiple comments attached to it, and comments can be associated with several elements, the association relation between the `Element` and `Comment` class has two asterisks to define these multiplicities.

**Query, Views, and Transformations** The OMG proposed the *Query, Views, and Transformations* (QVT) [44] standard for model transformation in the MDA process. QVT is not just one language, but a collection of languages specified as a MOF meta-model. The *Object Constraint Language* (OCL) [45]—a language to specify constraints and define query languages—has been adopted for querying models.

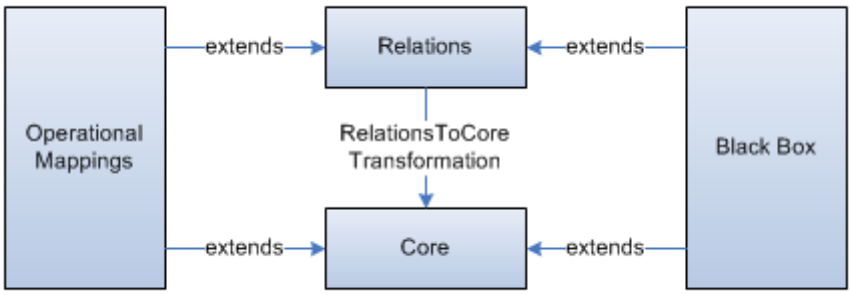



Figure 4.6: The QVT architecture

Next to OCL, QVT contains three languages for transforming models; these languages are called *Relations*, *Core*, and *Operational Mappings*. Both the Core and Relations language allow for specification of transformations between models in a declarative approach. The Relations language has, as opposed to the Core language, a graphical syntax and resides at a higher level of abstraction. Traceability, as defined in Section 3.1.3, is handled automatically in the Relations language, without involving the developer; in the Core language the traceability links are ordinary model elements, which have to be created by the developer. To convert a transformation specified in the Relations language into its Core counterpart, there is a transformation available, called the *RelationsToCore* transformation. The third language is the Operational Mappings language and extends the Relations language with imperative constructs—like loops, conditions, etcetera—and OCL constructs. The Operational Mappings language allows to use the declarative relation specification in an imperative approach. Added to these three languages is the *Black Box* mechanism, which supports executing external code during transformation execution. This way it is possible to develop a very complex transformation with a programming language of choice. An overview of the architecture of QVT is presented in Figure 4.6.

**ATLAS Transformation Language** Since QVT receives quite some criticism [65], we also present an alternative to QVT, the *ATLAS Transformation Language* (ATL) [30], which is not an MDA standard. The ATL architecture consist of three layers—as illustrated in Figure 4.7—namely *Atlas Model Weaving* (AMW), ATL, and the *ATL Virtual Machine* (ATL VM). ATL is the actual transformation language, but AMW can be used as a higher abstraction level transformation language. The ATL VM executes compiled ATL programs. We will not further elaborate on ATL as it falls out of the scope of this document; for more information of both QVT and ATL, and a comparison between the two transformation languages, we refer to [31].

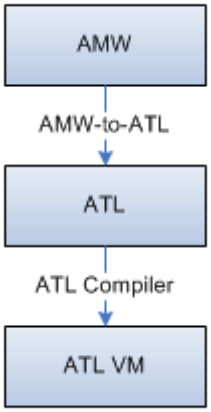



Figure 4.7: The ATL architecture

## 4.3 Comparing Software Factories and Model-Driven Architecture

Because we do not have the time to compare MDA and Software Factories based on a case study, we will only present the differences in the processes, models used, etcetera. We will not discuss which approach is better, has a higher efficiency boost, or results in the best maintainable source code, since we cannot base these statements on empirical results.

So, what are the differences between MDA and Software Factories? The major difference between the two is the specificity of the abstraction used in both approaches. Where MDA utilizes the general-purpose modeling language UML to raise the level of abstraction, Software Factories proposes DSLs. We should add that the approach taken in MDA is not necessarily general-purpose, because the OMG does offer the possibility to alter the UML to target at a specific domain; this can be done with UML Profiles. Choosing for a general-purpose approach will allow you to model each type of software system, but a more domain-specific approach will give a larger increase in reusability of components, and thus productivity [26]. A domain-specific approach is therefore preferred in the case we are dealing with a family of related products. This approach does require the effort to create a DSL—in the case a Software Factories approach is used—or a UML Profile—for the MDA approach—for the domain.

Another important difference between both approaches is platform support. Since software factories require the Visual Studio IDE for DSL usage, the approach is limited to the .NET platform. MDA can be applied on all platforms, as long as the programming language used is an object-oriented language. Although this is not an issue directly related to the software factories approach in software design, it will be a significant issue in practice.

Table 4.1 present a clear overview of the differences between Software Factories and MDA.

| | **Software Factories** | **Model-Driven Architecture** |
|---|---|---|
| **Organization** | Microsoft | OMG |
| **Supported Platforms** | .NET | All |
| **Modeling Language** | DSLs | UML |
| **Specificity** | Domain-specific | General-purpose |
| **Target** | Product families | No specific targets |
| **Tools** | Visual Studio and DSL Tools or GAT | A UML editor that supports the OMG standards and transformation tools |
| **Initial Efforts** | Creating the DSL and supporting tools | Creating a UML Profile to target a specific domain, if necessary, and specifying transformation rules |

Table 4.1: The differences between Software Factories and Model-Driven Architecture.

## 4.4 Summary

We have introduced and compared the two main approaches in MDD at this moment: Software Factories and Model-Driven Architecture. Both approaches use models, meta-models, and model transformations, but in different ways.

Software Factories build on software product line concepts to target product families. Through model-driven configuration of assets, a software factory allows to create each member in the product family. This configuration is done with so-called domain-specific languages; simple languages close to the problem domain that can either be in textual or graphical form. Software Factories increase productivity for development of product families, but it requires time and effort to setup the whole process and to create the needed DSLs.

MDA is a general purpose approach that allows to model each object-oriented software system, instead of only members in a certain product family. UML—and several other technologies—are adopted to model software systems independent of the target platform and to transform these models into platform-specific models, and finally source code. With UML Profiles it is possible to alter the UML to target more at a specific domain.

In [26] the authors claim that a higher specificity results in a higher increase in productivity; this implies that Software Factories have an advantage over MDA, if we only look at productivity and, of course, if this claim is valid. In contrast with Software Factories, applying MDA also result in increased portability—software systems are designed in a platform independent way—and better maintainability of software systems. The choice for one of both approaches depends highly on the target platform—Software Factories only support the .NET platform—and whether we are dealing with product families, or not.

# Chapter 5

# Mapping Model-Driven Development on the Problem Domain

In Chapter 2 we discussed the problem domain, defined what small and simple applications are, and introduced the concepts related to service-oriented architecture. Chapter 3 presented the basic theory of model-driven development. We have described the two main approaches—Software Factories and Model-Driven Architecture—and compared their main characteristics in Chapter 4. In this chapter we want to combine the knowledge of these chapters and present how we can apply the MDD approaches in the specified problem domain.

The main goal of this chapter is to explore the possibilities to apply a model-driven solution in the problem domain. This will help us in the decision for a specific application of MDD. We identify the spots in the small application we defined in Section 2.4 that we can target with our solution and define a list of criteria to check whether a given application of MDD fits our requirements. Making these spots and criteria explicit helps us in discussions with developers of Info Support's Professional Development Center (PDC) to choose the solution we will develop in the project assignment. The next chapter will define the actual project assignment.

This chapter is organized as follows. Section 5.1 identifies the spots in the defined small application where we can apply a model-driven solution. Section 5.2 lists the criteria we will use to validate a specific application of MDD; we will pay special attention to the effects on productivity, maintainability, and scalability. Section 5.3 contains three examples of applications of MDD that fit in our problem domain: model-driven service composition, model-driven UI navigation design, and rapid prototyping. We have added these examples to illustrate the possibilities of MDD in this domain. Finally, Section 5.4 summarizes this chapter.

## 5.1 Possible Locations to Apply MDD

In Section 2.4 we defined how a small application developed with Endeavour looks like. Now we want to identify the possibilities for applying MDD in such an application. We

have identified four application possibilities and listed these below.

1. *Apply MDD in a specific application layer.* We could apply MDD in one of the application layers—user interface, business logic, or data access—of an application. We could, for instance, target frontends to reduce the number of manually written LOC for user interface development.

2. *Apply MDD on data transfer between application layers.* In the previous application possibility we targeted a specific application layer, but we can also choose to look into the mapping between two layers in an application. One such mapping—for which a lot of frameworks already exist—is the object-relational mapping, which maps database records to business entities. If we look at how data is represented in each step from database to user interface, we get the following sequence:

   record→dataset→object→xml→object→user interface control

   We can apply MDD in each of these steps.

3. *Apply MDD to connect services.* In a SOA based on web services, services have to be specified in WSDL, registered in the registry, and be available for lookup for other services. Creating the connections between services could be done in an MDD approach, which is called model-driven service composition. A short introduction in model-driven service composition is given in Section 5.3.1.

4. *Apply MDD in a vertical approach.* Instead of targeting one specific layer or a connection between two layers, we could also research a vertical approach. We could, for instance, generate the database, business entities, and the object-relational mapping from a initially created UML model. An example of a vertical approach is the Ruby on Rails web framework[1], which generates business entities, object-relational mapping, controllers, and views from a given MySQL database[2].

With a clear view on the possibilities to apply MDD in our type of application, we have some guidance in choosing one specific application possibility later in this document.

## 5.2 Criteria for Validating an Application of MDD

To check whether an application of MDD is useful in our problem domain, we need a number of criteria for validation. We listed the criteria we will use to validate an MDD application below.

**Increase productivity** Since the main goal of our project is to increase productivity in developing small business applications, this is also our main criterium for selecting a solution.

[1] http://www.rubyonrails.org/
[2] http://www.mysql.com/

**Target small applications** Important for a solution is that it targets small applications, since this is the type of application we will be dealing with. A solution that is suitable for both small and large systems is of course welcome, but one especially for large applications will not do. We defined how a small application looks like in Section 2.4.

**Ensure maintainability** We are not looking for a solution that greatly improves productivity and at the same time results in a significant decrease of the maintainability of the software systems.

**Scalable to enterprise SOA application** It is important to design the resulting applications in such a way that they are still expandable to a full-blown enterprise SOA application.

**Applicable in a .NET environment** An obvious criterium is that we can apply it in a .NET environment, since Endeavour builds on this platform. Solutions that do not support the .NET platform will not be sufficient.

Above we only briefly introduced the issues related to the criteria for an MDD application. In the following three sections we want to give a more extensive description of the most important criteria. We start with productivity, followed by maintainability, and finally scalability.

### 5.2.1 Productivity

Above we stated that gaining an increase in productivity is the most important criterium in our project. How do we determine where we can best apply MDD to achieve the highest productivity increase? In the current development of software systems within Info Support all code is written manually. If we are able to generate a part of this manually written code, we can expect an increase in productivity. We should not forget that we need to design a model of the software system, before we can generate the source code. Defining this model will cost time too, which means that we cannot just calculate the percentage of generated lines of code to get the increase in productivity. In the worst case, the time to create the model takes more time than just writing the code manually. We should consider this fact if we draw conclusions on productivity increase. Another parameter that influences the productivity increase is the number of times we can apply our solution in the development process of a software system.

Think of a solution that reduces the amount of manually written lines of code for one mapping between two layers with a certain percentage. This will lead to a decrease in development time, since we have to write less code manually. The developer now specifies this mapping in a graphical editor, for instance, so we need to add the time needed to create the mapping in this editor. We can now determine the decrease—or increase, if defining the model is much harder than writing the code manually—in development time for each time we can apply the editor to specify the mapping. Say the decrease is rather small; does this mean that the solution is not interesting to look into? No, we now need to determine the

number of times we can apply this solution; if we have 50 possibilities to apply the solution, the overall increase in productivity may be very reasonable. The other way around, we may discover a solution that gives us a fairly large increase in productivity if we apply it a single time; if we can only apply the solution once in the development process, it might not be an interesting solution after all.

**Similarity Analysis** *Similarity analysis*—or *clone detection* in, for instance, [4]—is an activity to find similar pieces of code in the source of a software system. We can utilize similarity analysis to find similar manual lines of code that have a high occurrence in the software systems we are going to analyze in the analysis phase. These pieces of code may form a possibility for applying an MDD approach.

Simian[3] is an automated tool that can execute similarity analysis on a given set of source files. The tool supports Java, C#, C, C++, COBOL, Ruby, JSP, ASP, HTML, XML, and Visual Basic source code. Next to these programming languages, the tool can also analyze plain text files, like INI configuration files, batch files, and more. We may make use of Simian in future phases to do the similarity analysis on source repositories of several existing projects.

### 5.2.2 Maintainability

We should make sure that the applications developed with our solution are properly maintainable. In the best case we support bidirectional transformations as these allow the developer to synchronize between his model and code; he can choose to alter artifacts in both model and code. This has a positive effect on maintainability [58]. However, bidirectional transformations are harder to develop [29] and often not supported by existing tools [14].

Unidirectional transformations transform one or more source models into one or more target models, as we have seen in Section 3.1.3. If we decide to use this kind of transformations, we should determine where the developer applies changes if he wants to add functionality, for instance. We can choose for a solution where the developer creates a model, generates the code, and then leaves the model for documentation purposes only. One big disadvantage of this approach is that if the developer wants to change something in the design, he should alter the generated source code. That is not desirable because it would require the developer to completely understand the generated code, even if the code is badly structured and does not contain comments. A better solution is to separate generated code from manually written code in such a way that we can support regeneration of the generated code.

Since Microsoft released the .NET 2.0 platform, we can use partial classes to improve maintainability of software systems by separating generated from manual code. Below we introduce how partial classes work and how they give a gain in maintainability.

**Partial Classes** With *partial classes* we can spread class definitions over separate source files. On compilation of the source code, all partial definitions are gathered and combined

[3] http://www.redhillconsulting.com.au/products/simian/

```
public partial class Foo
{
        public void DoSomething()
        {
                /* Method body */
        }
}

public partial class Foo
{
        public void DoSomethingElse()
        {
                /* Method body */
        }
}
```

Listing 5.1: Definition of two partial classes (C#)

```
Foo foo = new Foo();
foo.DoSomething();
foo.DoSomethingElse();
```

Listing 5.2: Using the class `Foo` (C#)

into one class definition. Listing 5.1 illustrates how a class can be defined in two partial classes. The keyword `partial` is used to indicate that the class definition is a partial class. In the example we see a class `Foo` defined in two parts, with each part containing one method. When we compile and run this example we can create an instance of `Foo` and call both methods, as presented in Listing 5.2.

We can utilize these partial classes to separate generated from manual source code. This way we can hide the generated source code for the developer, like Microsoft did in their Windows Forms Designer in Visual Studio since .NET 2.0. Another advantage is that we can choose to regenerate source code from the model, as long as we do not change the method signatures. In the example in Listing 5.1, we could have generated the `DoSomething` method in the first partial class; if we do not change its signature, we can regenerate its body without breaking anything in the system.

### 5.2.3 Scalability

An important criterium is that we can expand the applications developed with our solution to enterprise SOA applications. To achieve this, we will design the solution in such a way that it produces applications with an architecture that is expandable to enterprise SOA. Because of this requirement, we will not choose for a client-server architecture with .NET Remoting, for instance. The applications developed with our solution will have their business logic in a business service and code for user interfaces is contained in a frontend. The

configuration items will not be tightly coupled to each other. With loose coupling we allow for multiple services to be added to the system at a later stage. We will also utilize web services for communication between configuration items, as this will be the case if the system is expanded.

## 5.3 Example Applications of Model-Driven Development

Below we will shortly discuss three examples of model-driven applications in the fields of service composition, user interface navigation design, and rapid prototyping, respectively. With these examples we hope to give the reader an idea on the possibilities of MDD in the specified fields. Note that we chose these fields based on the availability of papers and their relation to SOA.

### 5.3.1 Model-Driven Service Composition

One example of a model-driven approach in the SOA context is *model-driven service composition*. In Section 2.2.1 we introduced the concept composite service, a service that combines the functionality of several existing services. Service composition is the activity of composing one service from several available existing services. Doing this activity manually is a complex task and requires much knowledge, time, and effort [50]. A survey of research activities in the field of service composition—including model-driven approaches—is presented in [18].

We will now present two case studies that present an application of model-driven service composition.

Bézevin et al. present two MDA approaches for web service composition [8]. In the first approach, they model a PIM in UML and transform it to a PSM conforming to MOF compliant meta-models for Java, WSDL and Java Web Service Developer Pack (JWSDP). The transformations between PIM and PSM are specified in ATL, which we introduced in Section 4.2.3. In the second approach, both the PIM and PSM are based on the Enterprise Distributed Object Computing (EDOC) platform. The transformations are again modeled in ATL. Both approaches are illustrated with an example and analyzed in the concluding section of the paper. Of the two approaches, the second approach achieved better results, because EDOC provides elements closer to the problem domain and is designed for distributed systems, while UML is more generic. We can say that the more domain-specific approach performed better than the general purpose approach.

In [50], Orriëns et al. present a model-driven solution to service composition of existing web services, possibly offered by different providers. The authors advocate a phased approach where the developer starts with an abstract definition of the composite service and gradually makes it more concrete. The four phases—collectively called the service composition life cycle—are definition, scheduling, construction, and execution. The approach utilizes UML [46] for modeling service compositions and the Object Constraint Language (OCL) [45] for expressing business rules governing the composition process. An abstract meta-model—called the information model—defines the composition elements—activity, condition, event, flow, message, provider, and role—and relations between these elements

that can be used. We refer to [50] for a clear example—composing a travel plan service of existing flight ticket booking, hotel reservation, and car rental services—and a more extensive description of their approach.

We think that model-driven service composition is a very interesting research field; one that is really worth looking into. However, in this project we are targeting small applications which consist of a limited amount of services and use only a few—probably none—existing services. Therefore we expect that executing service composition in a model-driven fashion is not going to result in a major productivity increase for small application development. We do think that model-driven service composition can benefit Info Support in the development of enterprise applications.

### 5.3.2 Model-Driven UI Navigation Design

Developers nowadays have the luxury of a graphical designer to create their user interfaces, instead of having to do this by hand. This significantly increases productivity and is therefore indispensable in a professional development environment. However, if we have to implement navigation between different screens in a user interface we usually have to write the source code for this functionality ourselves. For larger software systems with many user interfaces, this can become a time consuming task. Therefore, research is done in the field of *model-driven UI navigation design*, where navigation between forms is modeled at a higher level of abstraction. We should note that we can distinguish between user interfaces for desktop applications—developed with Windows Forms or Java Swing, for instance—and user interfaces for web applications. Both types can be targeted with this solution, but may need a different approach.

A wizard is an example of an element in a user interfaces that needs navigational code to function properly. Wizards consist of a number of succeeding screens and their purpose is to ask input from a user in a user friendly approach; on the first screen you fill in some input fields, you press next to go to the second screen, you fill in the second screen, etcetera. Although it seems quite simple to implement this functionality, there are a few complications.

The first possibility you might think of is to create a collection containing the succeeding screens; each time the user presses the ‘Next’ button, the next screen in the collection is shown. If the current screen is the last screen in the collection, the ‘Finish’ button is shown instead of a ‘Next’ button. This approach might seem feasible, but wizards usually contain some conditional transitions between screens. Think of a wizard to add a car reparation to a software system of garage. The first screen of the wizard may give the user two options to choose whether the owner of the car that has to be repaired is an existing or a new customer. In the former case, you would want to select the customer from a list of existing customer and proceed. In the latter case the information of the new customer—like his name, address, bank account number, etcetera—has to be inserted first, so a screen to input these values into the system has to be shown first, before the information of the actual reparation will be inserted. Our initial solution will not work in this case, since we cannot deal with the conditional transitions. To support this kind of wizard we need a more sophisticated approach.

**The PetriX DSL** An example of a graphical DSL for navigation of a user interface in a web application is the PetriX DSL, presented in [41]. The given solution combines Petri Nets [52]—for modeling the interaction between forms—and XForms [10]—to specify the forms. In the PetriX context we talk about *partitions* and *interaction structures*. Partitions are the actual form specifications—the text boxes, drop down lists, labels, etcetera—and the interaction structures are the transitions between these partitions. Petri Nets, and therefore the PetriX DSL, support modeling of conditional transitions between partitions. The authors customized Microsoft Visio[4] to support PetriX diagrams. The developer first models the forms and interaction between these forms in Visio, using the graphical DSL. When the model is finished, the developer can export the model to an XML file, which is then passed to the XSLT-based transformation engine. The result of this transformation is XForms code. For a more thorough description and an example of the PetriX DSL, we refer to [41].

Since Info Support develops many software systems that require complicated wizards, they designed their own textual DSL based on XML to model relations between screens. At this moment this DSL does not have support for conditional statements, though it is possible to model the transitions in the DSL and specify the conditional statements in code. A more thorough specification of this DSL is not necessary at this point. It might be worthwhile to research how we can extend the Endeavour software factory with a graphical DSL supporting conditional transitions to increase productivity in development of frontends. In the next chapter we will see if model-driven UI navigation design is the best possible application to increase productivity for small application development.

### 5.3.3 Rapid Prototyping

The last application we want to discuss is categorized under the fourth possibility of the list we presented in Section 5.1. With *rapid prototyping* we mean a model-driven approach that allows us to quickly create a first working prototype of our system under development. This is a vertical approach, influencing all layers—user interface, business logic, and data access—in an application. Ideally, we start with a model in UML, for instance, and from that model we generate a database, stored procedures, data access code, business logic, and the user interface.

An example of such a vertical approach is the *Ruby on Rails* framework[5]. This is not a regular application framework, as discussed in [13] for instance. Ruby on Rails is a model-view-controller (MVC) framework for creating web applications that utilizes code generation to speed up the development process. The first step in the process is creating a MySQL database[6] for the web application. From the schema of this database, the framework can generate the model, controllers and views of your application. This technique is called *scaffolding*. The result of the scaffold operation is a working prototype of the system under development. Views are available for listing, adding, viewing, and removing data

[4] http://office.microsoft.com/visio
[5] http://www.rubyonrails.org/
[6] http://www.mysql.com/

and since the controllers are able to perform CRUD—create, retrieve, update, and delete—operations on the database, we have a working web application. Of course this will probably not satisfy all requirements, but the basis for the application is available without writing a single line of Ruby code. When the basis is there, the developer can replace the generated methods with his own methods.

If the next chapter concludes that a vertical approach will give us the highest increase in productivity, we could research a .NET alternative to Ruby on Rails. The question might raise whether we should invest in researching an alternative or just use the Ruby on Rails framework for our small applications. However, in the latter case we are violating the scalability requirement we elaborated in Section 5.2.3. Over time Info Support may decide to scale the small application to enterprise SOA and questions remain about scalability of Ruby on Rails applications [62]. Moreover, Info Support does not have any Ruby specialists available at the moment, and lacks experience and tool support in this field.

## 5.4 Summary

This chapter prepared us for the next chapter, where we will define the actual project assignment. We listed a number of criteria to help us decide between several applications of model-driven development in our domain. The most important criterium of this list is that the solution should increase productivity. We noted that the increase is not just the number of manual lines of code that we can now generate with our solution, but that we should also consider the time needed to design models. Two other important criteria are proper maintainability and scalability of the application.

To help us choose the best spot in a small Endeavour application to apply an MDD solution, we have identified four application possibilities for a model-driven solution. We can target a specific application layer or mapping between two application layers. A possibility more suitable for enterprise applications is to connect services in a model-driven approach. The last possibility is a vertical approach that generates code for all application layers from an initially created models.

We concluded this chapter we three examples to illustrate the power of model-driven development in our domain. Model-driven service composition is an interesting research field and can definitely support the development of enterprise applications, but is not suitable for the small applications in our project. The model-driven solution to design navigation in application frontends is definitely an application we will investigate in the next chapter. The same holds for the rapid prototyping example, which is an approach to quickly create a first prototype of an application.

# Chapter 6

# The Proposed Project Assignment

This document is only one deliverable in the overall project to design, implement, and validate a model-driven solution in the problem domain we described in Chapter 2. Now we have extensively investigated both the problem and solution domain, it is time to narrow the scope of our research and focus on a specific application part and model-driven approach to actually apply the gathered knowledge in the project assignment. This chapter presents two important decisions that we still need to make before we can actually develop a model-driven solution.

First we have to decide in which part of the application we will apply an MDD approach. We have identified the possiblities for locations to apply MDD in Section 5.1 and now we only have to choose one of them. Then we have to decide which specific MDD approach we apply. Do we follow the MDA or Software Factories processes as presented in Chapter 4, or decide to go for a custom approach. When both decisions are taken and justified, we can define the actual project assignment.

This chapter is organized as follows. In Section 6.1 we investigate metrics of past projects to choose an application part where we apply an MDD approach. Section 6.2 discusses the effect of the decision for an application part on the choice for a specific MDD approach. Section 6.3 defines the assignment for the project phase, where we will have to design, implement, and validate a model-driven solution in the problem domain.

## 6.1 The Choice for an Application Part

To find the best part of the application to apply our solution, we have investigated metrics of three past projects done by Info Support. All three projects satisfy the definition of small and simple application as presented in Section 2.4, and are developed with the .NET version of Endeavour. The metrics of Project 1, Project 2, and Project 3[1] are presented in Table 6.1, Table 6.2, and Table 6.3, respectively. The meaning of the columns in these tables is described below.

[1] Due to tangling concerns Info Support requested us to leave out the real project names.

| | |
|---|---|
| **Part** | The part of the application. |
| **LOC** | The number of lines of code (LOC) in the specified part. The lines of code contained in unit tests and code comments are not included in this number. |
| **MLOC** | The number of manual lines of code (MLOC) in the specified part. |
| **GLOC** | The number of generated lines of code (GLOC) in the specified part. |
| **%MLOC** | The percentage of LOC in this part of the application that is manually written. |
| **%Total MLOC** | The percentage of LOC of the total number of LOC that is manually written. |

There are a few numbers in the three tables that may need a little more explanation. The attentive reader might have noticed that for the first two projects the total number of LOC in the header of the table is equal to the number of LOC in the `Total` row, but in the third table these values—48.014 and 46.150, respectively—differ. The reason for this difference is that the third project contains 1.864 lines of unit test code, where the other two projects do not have unit tests. Since we are not aiming to generate unit tests in this project, we do not take into account the lines of code written for test purposes.

In the column `%MLOC` we present the percentage of the code that is manually written in the specified part of the application. This percentage can be calculated by dividing the value of the `MLOC` column by the value in the `LOC` column, and multiplying this value with 100%. If we, for example, calculate this value for the GUI part in the third project, we get the following calculation.

$$\%\text{MLOC} = \frac{\text{MLOC}}{\text{LOC}} * 100\% = \frac{18.374}{23.363} * 100\% \approx 79\%$$

Since each part can contain 100% manually written code, the values in the `%MLOC` column do not add up to 100%. Actually, the value in the `Total` row has no meaning for this column and is left empty.

The `%Total MLOC` column presents a percentage that is based on the total number of lines of code in the system, instead of the lines of code in this specific part of the application. For the GUI part in Project 3, this number is calculated as follows.

$$\%\text{Total MLOC} = \frac{\text{MLOC}}{\text{Total LOC}} * 100\% = \frac{18.374}{46.150} * 100\% \approx 40\%$$

What can we conclude from the metrics in these tables? For all three projects, most manually written lines of code are contained in the GUI. This gives us an indication that the GUI is the part of the application where we *might* achieve the best optimization. We emphasized the word 'might' because it is not guaranteed that this is the best place to apply MDD to reduce programming efforts. In the case that each manual line of code is so difficult that it cannot be captured in a model and has to be programmed manually, there is no opportunity to apply an MDD approach. This is of course an extreme example, but we should be careful with drawing conclusions from these metrics.

| **Project** | **Project 1**<br>16.868 LOC, 132 FPs, 860 hours | | | | |
|---|---|---|---|---|---|
| Part | LOC | MLOC | GLOC | %MLOC | %Total MLOC |
| GUI | 9.878 | 9.878 | 0 | 100% | ∼59% |
| Business | 3.368 | 3.368 | 0 | 100% | ∼20% |
| Data | 2.294 | 2.294 | 0 | 100% | ∼14% |
| Conversion / Batch | 1.328 | 1.328 | 0 | 100% | ∼8% |
| Total | 16.868 | 16.868 | 0 | | 100% |

Table 6.1: Metrics of Project 1

| **Project** | **Project 2**<br>19.588 LOC, 80 FPs, 700 hours | | | | |
|---|---|---|---|---|---|
| Part | LOC | MLOC | GLOC | %MLOC | %Total MLOC |
| GUI | 11.207 | 11.207 | 0 | 100% | ∼57% |
| Business | 2.425 | 2.425 | 0 | 100% | ∼13% |
| Data | 5.925 | 5.925 | 0 | 100% | ∼30% |
| Conversion / Batch | 1 | 1 | 0 | 100% | ∼0% |
| Total | 19.588 | 19.588 | 0 | | 100% |

Table 6.2: Metrics of Project 2

| **Project** | **Project 3**<br>48.014 LOC, 220 FPs, 1.594 hours | | | | |
|---|---|---|---|---|---|
| Part | LOC | MLOC | GLOC | %MLOC | %Total MLOC |
| GUI | 23.363 | 18.374 | 4.989 | ∼79% | ∼40% |
| Business | 7.051 | 7.051 | 0 | 100% | ∼15% |
| Data | 15.402 | 931 | 14.471 | ∼6% | ∼2% |
| Conversion / Batch | 334 | 334 | 0 | 100% | ∼1% |
| Total | 46.150 | 26.690 | 19.460 | | ∼58% |

Table 6.3: Metrics of Project 3

Although the metrics of past projects vote for an application in the GUI, we have decided to go for a vertical approach. The main reason for this choice is the current research done by the Professional Development Center (PDC) of Info Support. PDC is currently working on a new version of Endeavour that specifically aims to increase productivity. PDC knows, of course, that the productivity bottleneck for developing their software systems is the GUI layer, as the presented metrics prove. Therefore PDC is currently investigating possibilities to reduce programming efforts in frontend development. To achieve this reduction, they will look into DSL Tools and GAT—the software factory tools provided by Microsoft that we discussed in Section 4.1.4—and the applicability of frameworks.

To prevent that we do the same research in parallel and end up with two similar solutions, we decided to try for a more innovative approach. Innovative in a sense that we do not target one specific spot of an application, but try to generate a completely working application from an initially designed model.

## 6.2 The Choice for an MDD Approach

The choice for a vertical approach has a significant influence on the choice between MDA and Software Factories. Since we have to model a complete application instead of only one specific part in an application, we need a modeling language that allows us to specify enough information to generate all three layers—data access, business logic, and user interface—of the application. We need to be able to model business objects, data access objects, and views in a user interface. This directs us more to an MDA solution. We can of course develop a DSL that allows us to model business objects and their relations, but that would result in a UML type of language. Moreover, designing a DSL is a time-consuming and difficult job that requires expert domain knowledge [15]. In the time we have to complete this project, we do not think we can obtain enough domain knowledge to design and implement a DSL that covers all layers of an Endeavour application.

We want to make clear that the possibility offered by MDA to design software systems in a platform-independent way did not influence the decision between MDA and Software Factories. In Info Support's experience, porting a Java application to a .NET equivalent, or vice versa, never happens in practice, and therefore we will not take advantage of the portability benefit.

Now we have argued that MDA is the best choice for a vertical approach in our project, we should check if it fits the requirements for our specific problem. We have done an extensive search for tools and noticed that a vertical approach in practice usually generates code from a database schema rather than a UML class diagram. This does not directly imply that we can better use a database as initial model, but it triggered us to find out which of these two approaches is best in our project. Below we will discuss the two approaches more thoroughly and present their benefits and issues.

### 6.2.1 Generate from a UML Model

We can choose to precisely follow the MDA development process, as discussed in Section 4.2, where we create a PIM in UML, transform it to a PSM in UML, and finally generate the

C# source code. The MDA process is equivalent to the traditional object-oriented software development process, in a sense that both processes model the software system in UML before source code is written. This equivalence is an advantage, since UML is the language that is designed to model object-oriented systems [12]. Another advantage of UML to model in is that we are not restricted to generate applications that are connected to a database. Info Support can then decide to use the tool not only for data-driven applications, but also for applications that do not use a database at all.

Since we are targeting data-driven applications in this project, we should at some point create a database. It is common that the class model of software systems that are designed following the object-oriented development process does not directly map on the tables of the database used. The mapping between classes and database tables is often called object-relational mapping. If we want to generate a complete application, we should at least generate this mapping and maybe even the database. This is possible by annotating classes with a persistence attribute to indicate that objects of that class are saved in a database table [35]. This allows us to generate at least the object-relational mapping classes. Since [40] proposes to design databases with UML and [5] states that the transformation from UML model to database can be automated, we should have no problems in generating our database.

One issue is that due to the fact that most technologies used in the MDA process are not standardized, there is a lack of proper tools [29]. Since we want to generate code that is built on the frameworks and building blocks offered by Endeavour, we should have a tool that is customizable in such a way that we can achieve this. If such a tool is not available, we can use a template-based code generator like *CodeSmith*[2] or *MyGeneration*[3] to transform the XMI exports of a regular UML modeling tool into source code. We have already searched for existing templates that transform XMI to C# source code, with little result.

We have summarized the benefits and issues of UML as the modeling language below.

| **Benefits and issues of generating from a UML model** |
|---|
| (+) Follows the object-oriented principle |
| (+) Not restricted to data-driven applications |
| (+) Enough information to generate all application layers |
| (–) Few tools available |

### 6.2.2 Generate from a Database Schema

Another possibility is to first model and implement a database and then generate application source code from the database schema. This is similar to the approach used in the Ruby on Rails framework. It is clear that this is not an approach that matches the MDA or Software Factories approaches discussed earlier, but it is, in our opinion, a model-driven way of software development. This might require a little explanation. The most important view on a data-driven application is of course the view on the data to be managed. If we generate source code from a database, we can graphically design the database—most database management systems provide graphical database designers—and focus purely on the data.

[2] http://www.codesmithtools.com/

[3] http://www.mygenerationsoftware.com/

In other words, we raise the level of abstraction and omit all irrelevant details to design the most important aspect of the software system: data. This corresponds to the theory of model-driven development discussed in Chapter 3.

So how does this approach work? Given a relational database, we can obtain its schema and use it to generate source code for business entities, classes that interact with the database through SQL, and even views in the user interface. Business entities will directly relate to database tables and can therefore be generated given the schema of the tables. The access to the database will be implemented in separate classes that contain CRUD methods—create, retrieve, update, and delete—to manage data of business entities in the database. We have enough knowledge to completely generate these classes.

Generating code for the user interface will be quite a challenge, but we expect to be able to achieve good results here too. Given a table in a database, we can generate forms to add and edit business entities by mapping data types in the database to specific controls in the form. If we, for instance, have a column in the database of data type 'Text', we can generate a textbox control to edit the value in this column. Moreover, if Ruby on Rails is able to generate these views, we should be able to achieve the same results in a .NET solution.

Code generation can be implemented with one of the template-based code generation tools we mentioned above, CodeSmith or MyGeneration. For these tools there are several existing templates that generate business entities, data access code, and ASP.NET code from a database of almost any type. An example of such a set of templates is *.netTiers*[4] for CodeSmith. We can use one of these existing sets of templates or choose to develop our own templates that generate source code based on the frameworks offered by Endeavour. The availability of proper tools to implement this solution is of course a big advantage. It is also very helpful that there already exists a large number of templates that can generate several application layers and allows us to make a choice to use existing templates or build our own.

We have mentioned a few advantages—tool support and the possibility to generate code for all application layers—-of the database schema approach, but there is also one disadvantage. The existing templates all generate a business entity from a database table that is defined in the database schema. If, for instance, the database schema specifies a table `Customer`, the code generator will generate a C# class `Customer`. The `Customer` class will contain an attribute for each column of the `Customer` table. This implies that the class model of our software system is equivalent to the schema of the database. This is a disadvantage, since it violates the principles of good object-oriented design [12].

We have summarized the benefits and issues related to code generation from a database schema below.

| **Benefits and issues of generating from a database schema** |
|---|
| (+) Many tools available |
| (+) Enough information to generate all application layers |
| (–) Assumption that business entities directly map on database tables |

[4] http://www.nettiers.com/

## 6.3 The Project Assignment Description

Now we have presented all issues regarding model-driven development in a service-oriented environment, we can define a final project assignment. In the previous sections we have stated two choices: the application part where we will apply MDD and the MDD approach we are going to use. We have already chosen to research a vertical approach that targets all layers in the application, but the choice for an MDD approach is still to be made.

In the previous section we have discussed two possibilities for initial models in our MDD process—a UML class diagram or a relational database schema—and identified their benefits and issues. We now have to decide which of the two approaches we will apply in our project assignment.

Given the benefits and issues for both approaches and the fact that we have to generate data-driven applications, we have decided to go for the database as initial model. The only disadvantage for this approach—the assumption that the class model of the software system is equivalent to the database schema—is acceptable for Info Support. A big advantage of starting with a database is that we can use it to generate stored procedures, business objects, data access code, and even views in the user interface. With a UML diagram as starting point we can also generate all application layers, but have less tools available to implement our solution. Applying a code generation tool to transform the extensive XMI files outputted by a regular UML modeling tool into source code, stored procedures, and a database seems far more complicated than generating these files from a database schema. This feeling is supported by the fact that we have found numerous code generation templates for both CodeSmith and MyGeneration to generate code from a database schema, but had little result in our search for templates that take XMI files as input.

Although the decisions for these two choices defines our project assignment for a large part, we also want to elaborate a little on the target application—the application generated with our solution—and the way we will implement code generation. Then we will validate our solution to the criteria presented in Section 5.2 to see if our solution is expected to satisfy the requirements. Finally we will describe the validation phase of our project, where we will check if our solution really increased productivity in development of small Endeavour applications.

### 6.3.1 Target Application and Code Generation

Numerous times we have spoken about a vertical approach that can generate a complete application, but we have never defined precisely how this application will look like. This section will shortly define the target application and the tools and technologies used to implement code generation.

In Section 2.4 we presented how the architecture of a small application developed with Endeavour looks like. For our target application we will use the same architecture. This means that we have to generate a business service, with a business logic and data access component, and a frontend, as illustrated in Figure 2.2. The business service will be implemented as a web service to satisfy the requirement to expand to an enterprise SOA appli-

cation. A more detailed description of the target application will be given in the master's thesis.

The target application will be a .NET application; the language in which the source for the business logic and data access components will be generated is C#. We will use an SQL Server 2005 database to generate code from. With a template-based code generator we can access the schema of this database to generate C# source code for the business logic and data access components. In the data access component we will make use of the Data Access Building Block offered by Endeavour.

The front end of the target application will be a web application built on ASP.NET technology. The ultimate goal is to develop a generic solution that can generate both windows forms and web applications, but due to time constraints we have decided to focus only on web applications. A future project may research a generic solution.

We will make use of a template-based code generation tool like CodeSmith or MyGeneration to implement code generation. These tools provide functionality to read data from a database and generate code with the use of templates. We have searched for existing templates to generate a complete application and retrieved quite some results, but we decided to develop our own templates that generate code that builds on the building blocks offered by Endeavour. This will improve maintainability, since the developers from Info Support often work with Endeavour in their projects.

### 6.3.2 Is Our MDD Approach Worth Investigating?

We have identified a number of criteria in Section 5.2 to validate if a given MDD application is worth researching or not. We will now use these criteria to validate the proposed project assignment presented above.

**Increase productivity** We expect quite an increase in productivity since we will generate a very large part of the application. The goal is to completely generate the business service with its business logic and data access components. We do not have a clear view on what part of the frontend we can generate, but we expect to be able to at least generate forms to add, edit, delete, and list business objects. Added to the generation of C# source code, we can also generate stored procedures to communicate with the database. If we compare this with the original way of software development where we have to write all this code manually, we think our expected increase in productivity is justified.

**Target small applications** Our approach is able—and limited—to generate small applications. By choosing a database as initial model, we have assumed that the class model of the application is similar to the schema of the database it uses. For small applications Info Support can support this assumption, but for the large applications developed by Info Support it does not hold.

**Ensure maintainability** With our approach we are able to generate maintainable applications, but it is not a direct consequence. When we design the target application that is

generated with our solution, we should consider maintainability issues. There are several possibilities to improve the maintainability of the target application. Examples of these possibilities are the use of partial classes to separate manual from generated code and developing templates that generate code that builds on Endeavour building blocks.

**Scalable to enterprise SOA application** If we design the architecture of the target application in such a way that it corresponds to the Endeavour guidelines, we can ensure scalability of the application. We have presented the architecture of a small Endeavour application in Section 2.4; our target application will have an identical architecture. Implementing the business service as a web service, for instance, has a positive effect on scalability, as we can build several other business services that together form an enterprise SOA application.

**Applicable in a .NET environment** It is obvious that our solution is applicable in a .NET environment, since the code generation tools we have inspected all support C# code generation and access to the schema of an SQL Server 2005 database.

By validating the MDD application with the identified criteria, we have justified the choice for our project assignment.

### 6.3.3 Validating the Solution

Before we start the development of our code generation tool in the project assignment, we want to talk about validation. We can say that we think that our solution will improve productivity, but it would be far more valuable if we can prove this. Therefore we decided to include a validation phase in our overall project. We will validate our solution in three different ways, where we will focus on productivity, maintainability, and scalability. We will shortly describe each validation approach below.

1. *Empirical study.* As we have mentioned in the related work section of the introduction, a colleague at Info Support is working on a similar project to increase productivity for developing small Endeavour applications. Instead of model-driven development, he applies frameworks to reduce programming efforts. When both our solutions are finished, it would be an interesting experiment to find out which one results in the highest productivity increase. Therefore we will organize two sessions with two teams of two developers to compare our solutions. In each session the teams will reimplement a piece of an existing software system. The size of this piece will probably be one function point. In the first session team one will use the MDD solution and team two the framework solution; in the second session it will be the other way around. We will measure the time needed to finish the experiment and use this to compare our solutions. An overview of the experiment is given in Table 6.4.

2. *Metrics of past projects.* In the experiment described above we will let the teams implement a piece of an existing software system implemented by Info Support. We will choose a part for which Info Support has metrics available that show the time the

| | **MDD Solution** | **Framework Solution** |
|---|---|---|
| **Session 1** | Developer 1+2 | Developer 3+4 |
| **Session 2** | Developer 3+4 | Developer 1+2 |

Table 6.4: An overview of the empirical study.

developers needed to implement it. This will allow us to compare our MDD solution with the old approach in software development, and tells us if our solution actually had an improvement in productivity.

3. *Experts opinion.* The previous two validations both focused on productivity, but tell us nothing about maintainability and scalability. Since we do not have metrics available for these two quality attributes, we decided to use a subjective method for validation. Developers of Info Support with experience in developing small applications with Endeavour can probably tell us if our solution has a positive or negative effect on maintainability and scalability.

# Index

# Bibliography


[1] A.J. Albrecht and J.E. Gaffney. Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation. *IEEE Transactions on Software Engineering*, 9(6):639–648, 1983.

[2] Scott W. Ambler. Agile Model-Driven Development. Webpage (23 February 2007). `http://www.agilemodeling.com`.

[3] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley, 2nd edition, 2003.

[4] Ira D. Baxter, Andrew Yahin, Leonardo M. De Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone Detection Using Abstract Syntax Trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, pages 368–377. IEEE Computer Society, 1998.

[5] Sami Beydeda. *Model-Driven Software Development*. Springer, 2005.

[6] Jean Bézivin. In Search of a Basic Principle for Model Driven Engineering. *Upgrade*, V(2):21–24, April 2004.

[7] Jean Bézivin and Olivier Gerbe. Towards a Precise Definition of the OMG/MDA Framework. In *16th IEEE International Conference on Automated Software Engineering (ASE'01)*, pages 273–280, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[8] Jean Bézivin, Slimane Hammoudi, Denivaldo Lopes, and Frédéric Jouault. Applying MDA Approach for Web Service Platform. In *EDOC '04: Proceedings of the Enterprise Distributed Object Computing Conference, Eighth IEEE International (EDOC'04)*, pages 58–70, Washington, DC, USA, 2004. IEEE Computer Society.

[9] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web services architecture. W3C Working Group Note 11, W3C, 2004. `http://www.w3.org/TR/ws-arch/`.

[10] John M. Boyer, David Landwehr, Roland Merrick, T.V. Raman, Micah Dubinko, and Leigh L. Klotz. *XForms 1.0 W3C Recommendation*. W3C, 2nd edition, 2006. http://www.w3.org/TR/xforms/.

[11] A.W. Brown, S. Iyengar, and S. Johnston. A Rational Approach to Model-Driven Development. *IBM Systems Journal*, 45(3):463–480, 2006.

[12] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering – Conquering Complex and Changing Systems*. Prentice Hall, 2000.

[13] Xin Chen. *Developing Application Frameworks in .NET*. Apress, 2004.

[14] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *Online Proceedings OOPSLA03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, Anaheim, October 2003.

[15] Ahmet Demir. Comparison of Model-Driven Architecture and Software Factories in the Context of Model-Driven Development. In *Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MBD/-MOMPES06)*, pages 75–83, Washington, DC, USA, 2006. IEEE Computer Society.

[16] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[17] Arie van Deursen and Paul Klint. Little Languages: Little Maintenance? *Journal of Software Maintenance*, 10(2):75–92, 1998.

[18] Schahram Dustdar and Wolfgang Schreiner. A Survey on Web Services Composition. *International Journal of Web and Grid Services 2005*, 1(1):1–30, 2005.

[19] Mark Endrei, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, Pal Krogdahl, Min Luo, and Tony Newling. *Patterns: Service-Oriented Architecture and Web Services*. IBM Redbooks, 2004. http://www.redbooks.ibm.com/redbooks/pdfs/sg246303.pdf.

[20] Jean-Marie Favre. Foundations of Meta-Pyramids: Languages vs. Metamodels – Episode II: Story of Thotus the Baboon. In *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. http://drops.dagstuhl.de/opus/volltexte/2005/21.

[21] Jean-Marie Favre. Foundations of Model (Driven) (Reverse) Engineering: Models – Episode I: Stories of The Fidus Papyrus and of The Solarus. In *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany. http://drops.dagstuhl.de/opus/volltexte/2005/13.

[22] Franck Fleurey, Jim Steel, and Benoit Baudry. Validation in Model-Driven Engineering: Testing Model Transformations. In *Proceedings of 2004 First International Workshop on Model, Design and Validation.*, pages 29–40. IEEE Computer Society, 2004.

[23] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? Web Page. `http://www.martinfowler.com/articles/languageWorkbench.html`.

[24] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1st edition, 1999.

[25] Martin Fowler and Kendall Scott. *UML Distilled*. Addison-Wesley Professional, 3rd edition, 2004.

[26] Jack Greenfield and Keith Short. Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 16–27, New York, NY, USA, 2003. ACM Press.

[27] Jack Greenfield and Keith Short. Moving to Software Factories. *Software Development Magazine*, July 2004.

[28] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. *SOAP Version 1.2 Part 1: Messaging Framework*. W3C, 2003. `http://www.w3.org/TR/soap12-part1/`.

[29] B. Hailpern and P. Tarr. Model-Driven Development: the Good, the Bad, and the Ugly. *IBM Systems Journal*, 45(3):451–461, 2006.

[30] INRIA. *ATL User Manual v0.7*. `http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual[v0.7].pdf`.

[31] Frédéric Jouault and Ivan Kurtev. On the architectural alignment of ATL and QVT. In *Proceedings of the 2006 ACM symposium on Applied computing (SAC '06)*, pages 1188–1195, New York, NY, USA, 2006. ACM Press.

[32] Ümit Karakas and Sencer Sultanoglu. Software measurement. Web Page (19 March 2007). `http://yunus.hacettepe.edu.tr/~sencer/size.html`.

[33] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, April 2003.

[34] Charles W. Krueger. Introduction to Software Product Lines. Web Page (5 March 2007). `http://www.softwareproductlines.com/introduction/introduction.html`.

[35] Vinay Kulkarni and Sreedhar Reddy. Separation of Concerns in Model-Driven Development. *IEEE Software*, 20(5):64–69, 2003.

[36] Stephen J. Mellor and Marc J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley Professional, 2002.

[37] Stephen J. Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled: Principles of Model-Driven Architecture*. Addison-Wesley Professional, 2004.

[38] Tom Mens, Krzysztof Czarnecki, and Pieter van Gorp. A Taxonomy of Model Transformations. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. http://drops.dagstuhl.de/opus/volltexte/2005/11.

[39] J. Miller and J. Mukerji. *MDA Guide Version 1.0.1*. Object Management Group, 2003. http://www.omg.org/docs/omg/03-06-01.pdf.

[40] R.J. Muller. *Database Design for Smarties: using UML for data modeling*. Morgan Kaufmann, 1999.

[41] Martin Nussbaumer, Patrick Freudenstein, and Martin Gaedke. Web Application Development Employing Domain-Specific Languages. In *Proceedings of the 24th IASTED International Multi-Conference*, pages 13–18, February 2006.

[42] OASIS. *UDDI Specifications TC*, 2002. http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm.

[43] Object Management Group. *Meta-Object Facility*. http://www.omg.org/mof.

[44] Object Management Group. *Meta Object Facility (MOF) 2.0 QVT Specification*. http://www.omg.org/docs/ptc/05-11-01.pdf.

[45] Object Management Group. *Object Constraint Language*. http://www.omg.org/technology/documents/formal/ocl.htm.

[46] Object Management Group. *The Unified Modeling Language 2.0*. http://www.uml.org.

[47] Object Management Group. *UML Profile Specifications*. http://www.omg.org/technology/documents/profile_catalog.htm.

[48] Object Management Group. *XML Metadata Interchange (XMI) v2.1*. http://www.omg.org/cgi-bin/doc?formal/2005-09-01.

[49] Russ Olsen. Building a DSL in Ruby – Part I. Weblog. http://jroller.com/page/rolsen?entry=building_a_dsl_in_ruby.

[50] Bart Orriëns, Jian Yang, and Mike P. Papazoglou. Model Driven Service Composition. In *Service-Oriented Computing - ICSOC 2003*, pages 75–90. Springer Berlin / Heidelberg, 2003.

[51] Mike P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *WISE '03: Proceedings of the Fourth International Conference on Web Information Systems Engineering*, pages 3–12, Washington, DC, USA, 2003. IEEE Computer Society.

[52] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Technische Universität Darmstadt, 1962.

[53] Professional Development Center. *Architectuur – Logische Referentiearchitectuur*. Info Support, 2006. Version 3.0.

[54] Mauro Regio and Jack Greenfield. Designing and Implementing an HL7 Software Factory. In *Online Proceedings of OOPSLA05 International Workshop on Software Factories*, 2005.

[55] Genaina Nunes Rodrigues, Graham Roberts, Wolfgang Emmerich, and James Skene. Reliability Support for the Model Driven Architecture. In *Proceedings of the ICSE 2003 Workshop on Software Architectures for Dependable Systems*, pages 7–12, April 2003.

[56] Douglas C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[57] Ed Seidewitz. What Models Mean. *IEEE Software*, 20(5):26–32, 2003.

[58] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

[59] Shane Sendall and Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003.

[60] David Skogan, Roy Grønmo, and Ida Solheim. Web Service Composition in UML. In *Proceedings of the Enterprise Distributed Object Computing Conference, Eighth IEEE International (EDOC'04)*, pages 47–57, Washington, DC, USA, 2004. IEEE Computer Society.

[61] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML Models. In *UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools: 4th International Conference, Proceedings*, pages 134–148, 2001.

[62] Bruce A. Tate. *Beyond Java*. O'Reilly, 2005.

[63] Dave Thomas and Brian M. Barry. Model Driven Development: the case for Domain Oriented Programming. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 2–7, New York, NY, USA, 2003. ACM Press.

[64] Unknown Author. Domain-Specific Language Tools. Web Page. http://msdn2.microsoft.com/en-us/vstudio/aa718368.aspx.

[65] Unknown Author. QVT. Wikipedia.com (19 February 2007). http://en.wikipedia.org/wiki/QVT.

[66] Unknown Author. Turing Completeness. Wikipedia.com (14 February 2007). http://en.wikipedia.org/wiki/Turing_complete.

[67] W3C. *Extensible Markup Language (XML) 1.1 (Second Edition)*. http://www.w3.org/TR/xml11/.

[68] Hiroshi Wada, Junichi Suzuki, Shingo Takada, and Norihisa Doi. A Model Transformation Framework for Domain Specific Languages: An Approach Using UML and Attribute-Oriented Programming. In *Proceedings of the 9th World Multi-Conference on Systemics, Cybernetics and Informatics*, 2005.

[69] Web Services Description Working Group. *Web Services Description Language*. http://www.w3.org/2002/ws/desc/.