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

*Master's Thesis*

Maarten Schilt

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

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

SOFTWARE ENGINEERING

by

Maarten Schilt
born in Bergambacht, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Info Support BV
Kruisboog 42
Veenendaal, the Netherlands
www.infosupport.com

Cover picture: Launch of the NASA space shuttle Endeavour.

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

Author: Maarten Schilt
Student id: 1100432
Email: maartens@infosupport.com

**Abstract**

The Professional Development Center of Info Support noticed that the development process offered by their software factory Endeavour is too extensive for development of small applications. This project investigates how model-driven development (MDD) can contribute to increase developer productivity in developing this kind of application, without losing the current levels of maintainability and scalability. We discuss and compare approaches in MDD, and propose a solution that targets the applications in the scoped domain. The developed proof-of-concept transforms a database schema, together with a set of meta-data definitions, in a prototype of a web application. The generated source code is considered a starting point; custom functionality is implemented manually. We validate our solution by means of a case study and experts' opinions. The results are discussed, and a number of future research projects are proposed to target the weaker points in the proof-of-concept.

# Acknowledgments

This graduation project took me quite a lot of time and effort, and therefore I would like to thank the people who supported me during the project and contributed to the final outcome.

First of all, I would like to thank Andy Zaidman for supervising the project, and pushing me in the right direction when needed. Our discussions and your reviews of this document contributed greatly to the success of this project.

Info Support deserves gratitude for offering me the possibility to do this project. I especially would like to thank Dennis Joosten and Marco Pil for their supervision and advice, and Marieke Keurntjes for providing a joyful working environment. Finally, I would like to thank Martijn Beenes, Raimond Brookman, Patrick Gilhuijs, and Marcel de Vries, for participating in the validation phase of the project.

I would like to thank my friend, future colleague, and fellow graduation student, Bastiaan Pierhagen for his advice, the unavoidable framework-versus-modeling discussions, and numerous reviews of this master's thesis.

On a personal note, I would like to thank my friends, and my girlfriend Ilse in particular, for their love and support during the project. Last, but not least, I owe thanks to my parents for offering me the opportunity to study, and for always believing in me.

# 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 [28]. *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 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.

Within the Professional Development Center of Info Support the question raised how model-driven development could benefit the development process of small applications. The development process for these applications defined in their software factory Endeavour is too extensive, and needs an increase in productivity. This led to the formulation of an assignment for our graduation project: research the applicability of model-driven development for small Endeavour application development with the goal of increasing developer productivity. This thesis describes the outcomes of this research project.

## 1.1 Document Overview

This thesis is organized as follows. Chapter 2 gives an extensive definition of the project, and the problem we try to solve. We present a problem domain analysis in Chapter 3. This is followed by two chapters that are based on available literature in the field of model-driven development; Chapter 4 discusses the basic concepts of MDD, and Chapter 5 describes the two main MDD approaches: Software Factories and Model-Driven Architecture. With the knowledge of the first four chapters, we narrow the scope of our research project and formulate a solution direction in Chapter 6. Chapter 7 identifies the criteria we used in the validation phase of the project. A description of the proof-of-concept that we developed in the implementation phase, is given in Chapter 8. Then Chapter 9 describes a case study to which we applied our solution, and presents the benefits and issues we encountered during this study. For validation we also organized a session where domain experts gave their opinion on our approach and solution; the results are given in Chapter 10. Finally, we draw conclusions and identify future work in Chapter 11.

# Chapter 2

# Project Definition

In this chapter we present the details of the graduation project we carried out at Info Support. We first describe the overall goals and obtain an overall research question from these goals. Then we divide the research question into small chunks that will be dealt with in four separate project phases. We also introduce research related to this project.

The chapter is organized as follows. Section 2.1 describes the context in which this project is performed. Section 2.2 identifies the main goals of the project. Section 2.3 presents the project's research question, and the sub questions in which the overall question is divided. Section 2.4 distinguishes phases in the project and identifies the chapters in this thesis that belong to each phase. Section 2.5 contains related work.

## 2.1 Project Context

The project is carried out at Info Support as a master's project under supervision of the Software Evolution Research Lab. Both parties are briefly introduced in this section. We pay special attention to the Professional Development Center (PDC) of Info Support, as this is the department where our assignment originated from.

### 2.1.1 Software Evolution Research Lab (SWERL)

The graduation project is supervised by the Software Evolution Research Lab (SWERL). SWERL is part of the Software Engineering Research Group, department of Software Technology, faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology.

SWERL aims to address deterioration of the structure of evolving software systems by performing research in three different directions. Firstly, they try to develop new technologies that help prevent this deterioration when changes to a software systems are made. Secondly, they research program analysis techniques to uncover parts of software systems that continuously evolved in their life cycle. Finally, they investigate restructuring techniques for complex software systems.

### 2.1.2 Info Support

Info Support is an IT service provider with approximately 250 employees in the Netherlands and another 50 in Belgium. The services offered to their clients are training, software development of enterprise administrative applications, life cycle management, and hosting. Clients of Info Support are mainly top 500 organizations in the Netherlands and Belgium in the markets industry, finance, health-care, and government.

Info Support presents itself as Solid Innovator, aiming to develop solid and innovative software solutions to support organizations in realizing their business goals. To achieve innovation, Info Support constantly gathers technical knowledge and shares this among their developers to be applied in their projects. An important department of Info Support that facilitates this process is the Knowledge Center that organizes trainings for both internal developers and external parties.

### 2.1.3 Professional Development Center (PDC)

The Professional Development Center (PDC) of Info Support is responsible for ensuring a professional approach to software development within Info Support. To realize this goal, PDC developed the software factory Endeavour that applies structure and standardization to the software development process of both .NET and Java applications. Endeavour offers a RUP-like development process, tools, standards, guidelines, logical and technical reference architectures, building blocks and training. With their 'embrace and extend' policy toward new technologies, Info Support constantly updates Endeavour with cutting-edge technological solutions.

## 2.2 Project Goals

Although Endeavour proved itself in the development of enterprise applications, the PDC noticed that the process followed with their software factory is too extensive for the somewhat smaller and simpler applications. Therefore the PDC decided to investigate state-of-the-art technologies to increase productivity in developing these small and simple applications without losing the quality level achieved with the original Endeavour approach. The assignment and main goal for this graduation project is to research one such technology: model-driven development. Comparing model-driven development with other productivity increasing technologies is not a goal, as it is not in the scope of this project.

## 2.3 Project Research Question

In the previous section we defined that the main goal of this project is to research whether model-driven development is a good candidate technology to be applied to develop small and simple applications. The technology is considered 'good' if it increases the productivity in developing these applications without deteriorating their quality attributes.

From this goal we have formulated the following research question. The remainder of this thesis answers this question.

> *How can we apply model-driven development to increase productivity in developing small and simple Endeavour applications without losing the quality level achieved with the original approach?*

Because it is hard to answer this question as a whole, we decided to split it up in sub questions. With the answers to all these sub questions we hope to be able to answer the overall research question in the conclusion of this thesis. In the succeeding chapters the following sub questions are answered.

**Chapter 3** *What does the problem domain look like?* How are small and simple applications defined? What is service-oriented architecture? What is Endeavour?

**Chapter 4** *What is model-driven development?* What are models, meta-models, and transformations? What are the known benefits and issues of applying model-driven development?

**Chapter 5** *What approaches are available in model-driven development?* What do these approaches offer and how do they compare?

**Chapter 6** *How can we apply model-driven development in our domain?* Where and how do we apply model-driven development? What tools do we use?

**Chapter 7** *What criteria are needed to validate our solution?* Why are they needed?

**Chapter 8** *What is our solution?* How is it implemented? How did we deal with the identified criteria? What are the limitations of the solution?

**Chapter 9** *How can we apply our solution?* What involves using the solution in a project? What are our first impressions on benefits and issues?

**Chapter 10** *How did our solution perform?* Did the productivity increase? Did we achieve a quality level comparable to the original development process? What activities are needed to better satisfy the identified criteria?

## 2.4 Project Phases

If we look at the sub questions of the previous section, we can divide the project in four phases. In the list below we describe these phases and state the chapters that belong to each phase.

**Research phase** The project is initiated with a research phase to obtain proper knowledge in the field of model-driven development. In this phase we read several papers about the theory, approaches, and applications of model-driven development, and explain the relevant knowledge in this thesis. (Chapter 3, 4, and 5)

**Analysis phase** In the analysis phase we try to find out how model-driven development fits in our domain. At the end of this phase we have decided the solution we will implement in the implementation phase. We also identify the necessary criteria to validate our solution when it is finished. (Chapter 6 and 7)

**Implementation phase** The actual implementation of the model-driven solution is done in the implementation phase. (Chapter 8)

**Validation phase** In the validation phase we will validate if our solution improved productivity compared to the original approach in software development, and if the quality of the resulting applications is not significantly deteriorated. (Chapter 9 and 10)

Although the chapter categorization suggests a strictly sequential process, we do expect overlap between the four phases. During the research phase we probably process the already gathered knowledge and analyze how that part could help us in the specified domain. Furthermore, in our choice for a tool in the analysis phase it is likely that we perform a few test cases with the tools, that may form the basis for the following implementation phase. There are probably more occasions where phases overlap during the project.

## 2.5 Related Work

While we research model-driven development to reduce programming effort in developing small and simple Endeavour applications, a colleague and friend of mine, Bastiaan Pierhagen, researches the applicability of frameworks to achieve the same goal. Our projects are performed completely individual, but we do share knowledge and discuss related topics along the way. A comparison between our two solutions is not in the scope of our project, but is considered future work.

# Chapter 3

# Exploring the Problem Domain

To find a proper solution for the problem presented in the previous chapter, we should have enough knowledge of the domain in which this problem resides. Because the central research question explicitly states that we target small and simple Endeavour applications, we investigate and define this type of application in this chapter.

The chapter is organized as follows. In Section 3.1 we give a definition of a small and simple application in this project. Then we introduce Info Support's software factory Endeavour in Section 3.2. Finally, we explain the architecture of a small and simple application that is developed with Endeavour in Section 3.3.

## 3.1 Small and Simple 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) of its source code. 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 [33].

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 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

| **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 3.1: Comparing LOC and FPs for expressing the size of a software system

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 [33]. We have summarized the advantages and disadvantages of measurement with LOC and FPs in Table 3.1.

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 define a small application as an application that consists of up to 300 function points.

A simple application in this project is an application that contains little business logic and has the sole purpose to manage data in an underlying database. This kind of application offers CRUD—create, retrieve, update, and delete—functionality to its users through a graphical user interface.

## 3.2 Endeavour

Info Support created the *Endeavour* software factory for professional development of enterprise business applications. Endeavour provides developers with a highly-structured process to develop administrative applications with a service-oriented architecture. With a good mix of people, processes, and tools, Endeavour aims to achieve high software quality and precise estimation of project duration.

We do not aim to provide a full description of the features offered by Endeavour, but rather work toward a clear view on the architecture of applications that are developed with Endeavour. Therefore we first introduce the concept that highly influences the architecture of Endeavour applications, namely service-oriented architecture.

### 3.2.1 Introduction to Service-Oriented Architecture

As the name implies, a *service-oriented architecture* (SOA) is an architecture based on services. But what exactly is a *service*? We first answer this question, before we look into the SOA concept.

**Services** Services are highly-coherent software components that can perform one or more functions. These functions are defined in a *service description* and implemented in the so-called *service implementation*. 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 service implementation is the component offering the functionality specified in the service description.

An example of a service is a customer management service to manage all customers of an organization. This service may provide a `MoveCustomer` function, which is called when a customer moves from one address to another. The service description of this service defines the signature of this function and the input and output messages. The input message probably contains the identifier of the customer we have to move, and his new address. When the function is done, the output message can describe whether the operation was successful or yielded a few exceptions during its execution, for instance.

Functions of services are related to business processes [51]. The design of a service therefore starts with identifying the business process it should implement. In the example above we targeted the business process of moving a customer, and proposed a `MoveCustomer` function to implement this process. We could have suggested a more generic `UpdateCustomer` function that saves all fields of a given customer—including the address field—but this would violate the principles of service-oriented design.

According to Papazoglou, services have three important characteristics [52], 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.

Finally, we want to introduce two types of services that we encounter later on in our description of the Endeavour software factory, namely composite services and web services.

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 call the travel booking service, without knowledge of the three underlying services.

A popular implementation of a service is the *web service* [7]. The service description of web services is specified in the *Web Service Description Language* (WSDL) [67], which is a language based on XML. Messages used for communication with web services are *Simple Object Access Protocol* (SOAP) messages [27]. *Universal Description, Discovery, and Integration* (UDDI) [42] is the repository used for publication and discovery of web services. Note that implementing services does not require web service technology, and that applying web service technology does not imply good service-oriented design [51].

The service characteristics already hinted that it is common to have multiple services that together form a software system. The remainder of this section describes what happens if we build an architecture of several interacting services.

**Service-Oriented Architecture (SOA)** A SOA consists of multiple services interacting with each other through a standardized communication protocol. Three important types of participants in a SOA are the *service provider*, the *service registry*, and the *service client* (or *service consumer*).

The service provider is the person or organization offering a service, and is expected to publish the description of this 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, a client can invoke the functions offered by the service. An overview is given in Figure 3.1.

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 Info Support implements SOA with web service technology, this is not necessary. A SOA can also be implemented using, for instance, Java Remote Method Invocation (RMI) or .NET Remoting. However, these two technologies do not allow for communication between services running on different platforms.

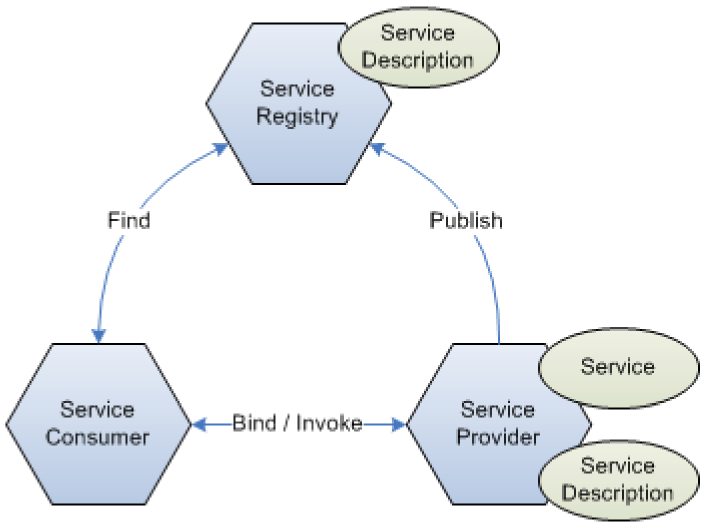


Figure 3.1: Service-Oriented Architecture [18]

### 3.2.2 SOA in Endeavour

The architecture of applications developed with Endeavour inherit from pre-defined logical and technical reference architectures, that incorporate the concepts of service-oriented architecture. The reference architectures identify six different types of components, called *configuration items* [53]. Some configuration items are related to core business functionality, while others are used to solve a technical problem. A short description of each type of configuration item is given below.

**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.

**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 3.2.1.

**Integration Service** The *integration service* connects an external system to the internal communication standard for, for instance, business-to-business 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 a service implementing authorization for the application.

**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) and ASP.NET (for .NET 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 3.2.1. 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.

A typical enterprise application developed with Endeavour consists of multiple business services, process services, integration services, platform services, and frontends, interacting with each other through a service bus. For small applications we do not need all these types of configuration items, as we explain in the next section.

## 3.3 Small and Simple Applications in Endeavour

From the list of configuration items in the previous section, only one business service and one frontend are used in a small Endeavour application. The business service contains all business logic of the application, and maintains a connection with the underlying database. The interaction with the user is the responsibility of the frontend. Figure 3.2 illustrates the architecture of a small and simple application.

Because the application contains only one service, it is not necessary to include a service bus in the architecture. Instead of using a service locater—a component corresponding to the service registry as introduced in Section 3.2—we directly connect the frontend to the business service. Expanding the system with new services requires us to include a service bus, and use a service locater for discovery of services.

What we do need is a contract for the business service, which is divided in a service contract and a data contract. The service contract defines the functions that are implemented by the underlying service. The messages that are exchanged in calling one of these functions, are defined in the data contract.

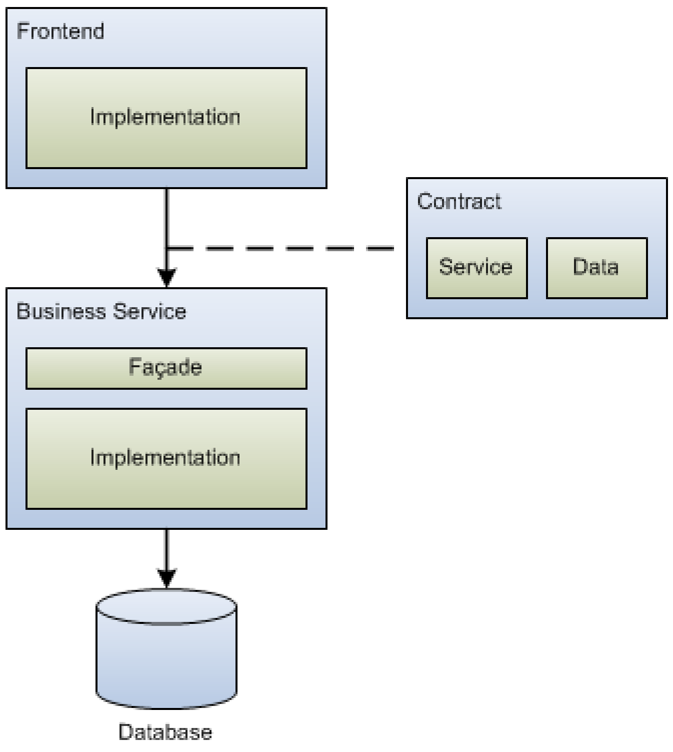


Figure 3.2: A small SOA application in Endeavour

# Chapter 4

# 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 [28]. 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 4.1 presents the basic concepts of MDD: the model, the meta-model, and the model transformation. Section 4.2 deals with the expected effects of MDD on software quality attributes. Finally, the benefits and issues of MDD are discussed in Section 4.3.

## 4.1 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.

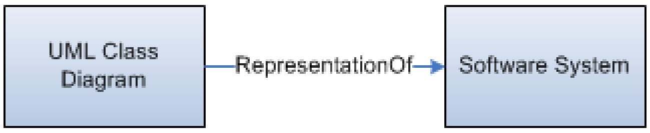


Figure 4.1: A UML Class Diagram is a model of a software system.

### 4.1.1 Models

In literature, several definitions for the concept of *model* exist [6, 40, 57]. Each of these definitions mentions a system and a relation—called the *RepresentationOf* relation [20]—between the system and a model. The system is represented by a model, but it does not have to be a full description. A model is created with a particular goal in mind and therefore only information necessary to achieve this goal is included in the model [6]; 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 4.1.

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 be any real-world object, like a building or a bridge. From now on we 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 now look at characteristics that determine whether 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 listed these attributes below.

**Abstraction** Through abstraction a model hides 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 get a better overview on the system's functionality, which results in higher quality and productivity [28]. 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 [28]. Abstraction is the most important attribute of a model [58].

**Understandability** Just hiding 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, while 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, because this abstraction does 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.

At a later stage in this project we use these quality attributes to validate whether the chosen model is a good representation of the application described in Section 3.3.

**Classification of models** Models can be classified using different approaches. One such approach is to make a distinction between *specification models* and *description models* [40, 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 focus on forward engineering and therefore deal with specification models.

Another classification approach is to distinct between *sketchy models*, *blueprint models*, and *executable models* [24, 38]. Sketchy and blueprint models can be used 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 of how an existing system works, again without all the details. Sketchy models aim at selective communication instead of complete specification [24].

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 [24].

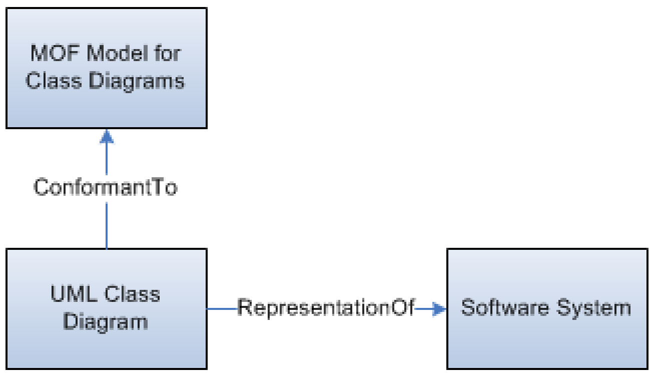


Figure 4.2: The UML Class Diagram conforms to the MOF Model for Class Diagrams.

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. The use of executable models is not widely spread [20]. Executable UML is an approach for executable models [37].

### 4.1.2 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.*

If we summarize this definition, we can say that a meta-model is a model of a set of models [19]. The models in this set are all specified in a certain modeling language. For example, the UML meta-model is a model for all models that are specified in the UML. The relation between a valid model and its meta-model is called the *ConformantTo* relation [5, 19]. We expanded Figure 4.1 with the ConformantTo relation and the UML meta-model, which results in Figure 4.2.

We illustrate the concept of meta-model with an example containing an excerpt of the UML meta-model for class diagrams. The UML meta-model specifies which constructs we are allowed to use in these diagrams. We presented the excerpt in Figure 4.3.

The meta-model defines that a class has a name and 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 class, modeled with the `from` and

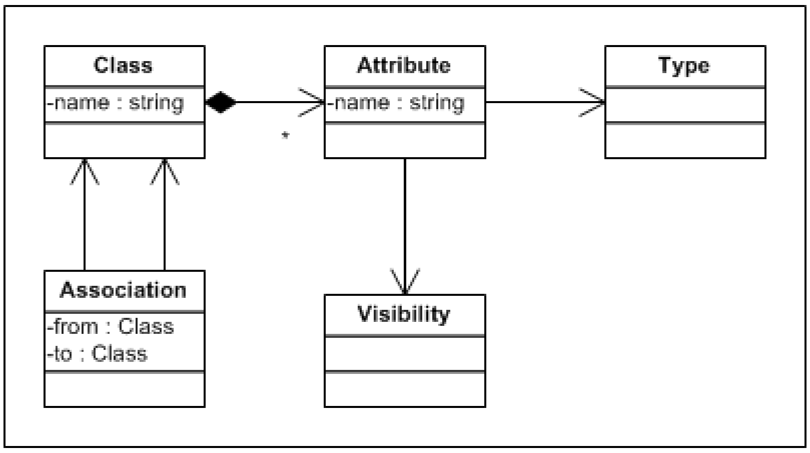


Figure 4.3: Example of a piece of the UML meta-model in UML

`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 already mentioned that the meta-model is a specific kind of model, therefore a meta-model can also be described in a modeling language. In the example above we used UML to define the UML meta-model for class diagrams. The Object Management Group proposed the *Meta-Object Facility* (MOF) [43] for meta-modeling, and used this language to describe the meta-model of UML models.

There are three reasons for existence of meta-models [38]. Firstly, a meta-model specifies the language you are modeling in, as we have seen in the definition 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, as we will see in Section 4.1.3.

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.

### 4.1.3 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. At the same time, these transformations between models can become very complex [39], especially at higher levels of abstraction.

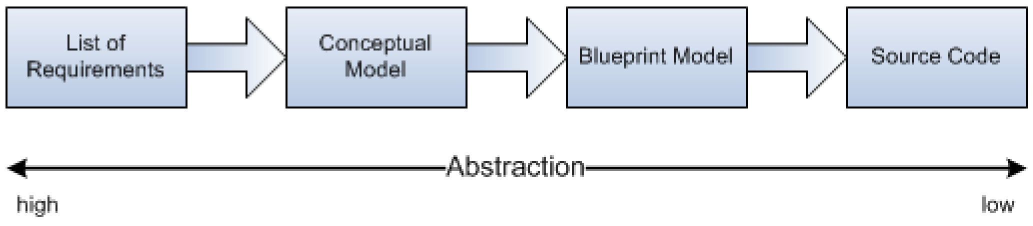


Figure 4.4: Example of models and transformations in the MDD process.

In the introduction of this chapter we described the traditional 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, as illustrated in Figure 4.4. The process starts with a model at a very high level of abstraction—the requirements—and transforms these into models of lower level of abstraction, till we end up with a working software system. In this section we give an overview of common characteristics of model transformations, and indicate why these transformations are sometimes very difficult to define.

Model transformations take at least one model as input to produce at least one output model, but it is also possible to have multiple input or output models [39, 59]. Meta-models of the input and output models are needed to specify transformations between two models; this is illustrated in Figure 4.5. The transformation between the source model and target model is defined in a number of transformation rules that make use of the meta-models of both the source and target model. These rules are specified in a transformation language and conform to a certain meta-model. The transformation rules are therefore also a model, not of the software system, but of the transformation between a source and target model.

Let us take a look at a transformation from a UML class diagram to C# source code, to illustrate the generic description of model transformation above. The transformation rules for this transformation describe how an arbitrary UML class diagram is transformed into C# code. Such a rule defines, for instance, that a class node in the UML model results in a class definition in the source code. Another rule might specify that a UML class attribute is transformed in a C# class attribute and a C# class property.

Figure 4.6 illustrates these transformations for a UML class diagram that contains a `Customer` class. The `Customer` class in the class diagram is transformed in a `Customer` class declaration in the source code. The attribute and property declarations in the C# source code are the result of the `name` attribute in the UML diagram.

With a global idea on what transformations are capable of, we now present a few important characteristics of model transformations, namely how transformations are executed, what kind of models are involved in the transformation, and model transformation directionality. We also present the required and desired characteristics of model transformation languages and tools.

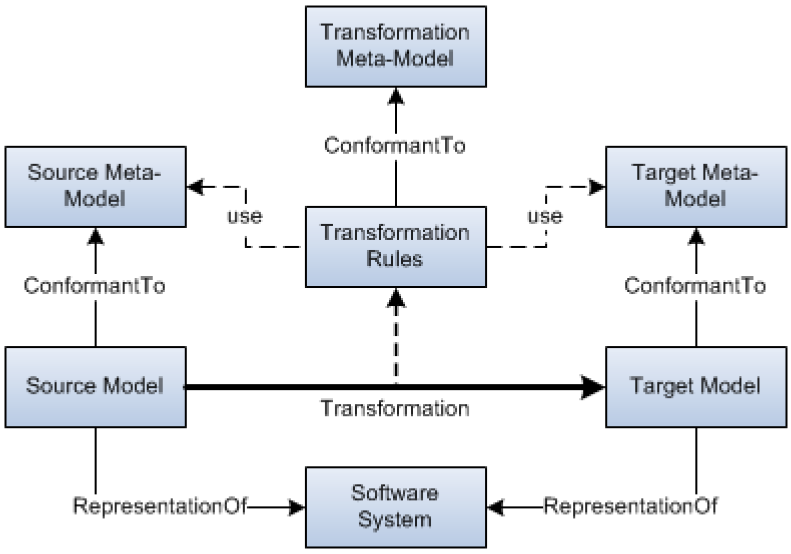
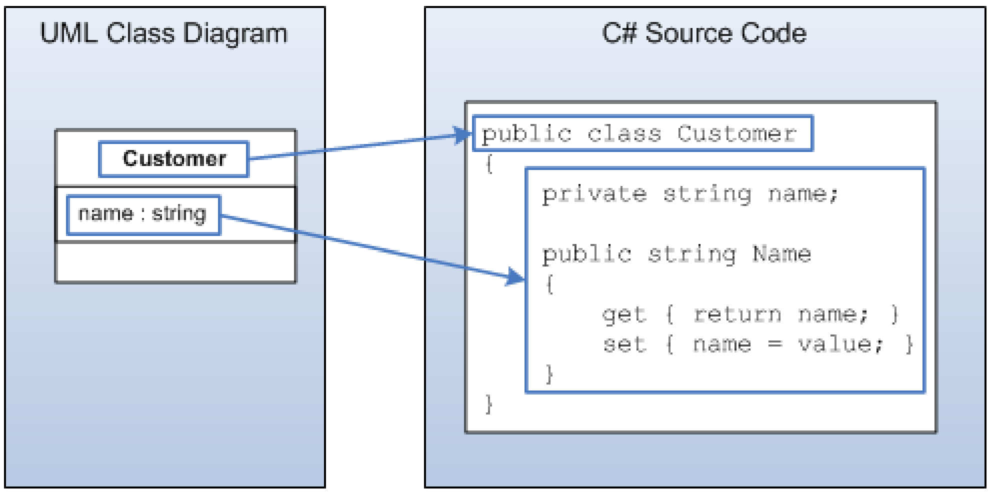


Figure 4.5: The model transformation.



```
public class Customer
{
    private string name;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```

Figure 4.6: Transformation from a UML class diagram to C# source code.

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

Listing 4.1: An attribute to indicate a unique identifier (C#)

**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 semi-automatic transformations is to *mark* models with meta-data [8, 14, 28]. Extra information, unnecessary for the viewpoint on the system or the level of abstraction, is added to the model to make model transformations possible.

In UML it is, for instance, possible to mark classes or attributes with so-called stereotypes. If we would like to specify that an Integer attribute of a certain class represents a unique identifier, we could use a `<<UniqueId>>` stereotype to define this. The tool that executes the transformation to source code sees this mark and generates other code than it would have when no stereotype was specified.

Marking in code can be done by adding attributes to programming elements [66]. In Java these attributes are called annotations; C# calls them attributes. An example of attribute usage is given in Listing 4.1. Attributes can also be used to mark business entities, persistent objects, methods that require logging, and much more.

Manual transformations are also possible but not desirable, because they are more sensitive to faults and require more effort. A transformation that is commonly performed manually is the transformation from a list of requirements to a conceptual model of the software system, as the source model in this case is specified in natural language and is often ambiguous, inconsistent, and incomplete [39].

**Transformation type** We can distinguish between *model-to-model*, *model-to-code*, and *refactoring* transformations [8, 14, 39]. A model-to-model transformation converts a model into another model, like a transformation from a UML class diagram to a database 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 [39, 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 producing 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 [23]. To illustrate refactoring transformations we give an example of the Extract Method refactoring operation on source code.

```
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 4.2: A method to print order information (C#)

```
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 4.3: The refactored version of Listing 4.2 (C#)

In Listing 4.2 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 4.3. 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* [39]. 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

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

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* [28]. With a unidirectional transformation 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 [28]. 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 do not yet discuss specific transformation languages and tools, but we give an overview of the characteristics these languages and tools should possess to be applicable. There are also a few characteristics that are not required, but desirable. In Chapter 5 we introduce the transformation languages ATL and QVT.

Sendall and Kozaczynski state that transformation languages must be expressive, unambiguous, and Turing complete [59]. 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 [39]; 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 [39].

**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 [21].

## 4.2 Effects on Software Quality Attributes

We 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. In the literature we gathered in this project we found several influences of MDD on software quality attributes. We describes these influences in this section.

As stated in the ISO 9162 standard, software quality is characterized by the following six attributes [31].

**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 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 [28]. 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. A good example of an MDD approach that contributes to system portability is Model-Driven Architecture, which we elaborate on in the next chapter.

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 always correctly represent the system [34]. In the case unidirectional transformations are used, the system is generated from the source models, and no changes to the models are made automatically when 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 is suitable for achieving a high reliability for the developed system.

## 4.3 Benefits and Issues

Before one decides whether to apply a new software engineering approach or not, it is important to know the resulting benefits and issues. We 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 write 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 [8, 28], as we have seen in the previous section.

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 [28]. Since current platforms and frameworks are getting more and more complex [56], this reduction is most welcome. The raise of abstraction level

| **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 4.1: Benefits and issues of model-driven development

also results in a vocabulary closer to the problem domain [28, 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? In our introduction to model transformations, we 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. These transformations will probably always be executed manually.

Hailpern and Tarr question whether MDD reduces complexity or just moves complexity from development of models to the model transformations [28]. 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 [28]. 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 [28]. 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. Next to extra effort, this also requires expert domain knowledge.

In Table 4.1 we have summarized the benefits and issues related to MDD.

# Chapter 5

# 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 presents 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 proposed the UML to create models in, SF build on domain-specific languages (DSLs). We elaborate on both approaches and their ways of software design.

Beside Model-Driven Architecture and Software Factories, there are other, less popular approaches in MDD, namely Domain-Oriented Programming [62] and Agile Model-Driven Development [2].

This chapter is organized as follows. Section 5.1 discusses Microsoft's Software Factories. Section 5.2 introduces Model-Driven Architecture from the Object Management Group. Section 5.3 gives a comparison of both approaches, based on characteristics.

## 5.1 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 [25], 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 answer these questions and explain how MDD is applied to realize software factories.

The remainder of this section is organized as follows. Section 5.1.1 describes the dimensions of innovation on which software factories build. Section 5.1.2 elaborates on domain-specific languages, the languages in which models of a software system are defined for a specific domain. Section 5.1.3 gives a detailed explanation of how software factories work. Section 5.1.4 introduces tools available for development of software factories.

### 5.1.1 Dimensions of Innovation

Software Factories build on three main dimensions of innovation—abstraction, granularity, and specificity—as presented in [25]. To innovate along these dimensions, SF makes use of two development methodologies, *model-driven development* (MDD) and *component-based development* (CBD). We now discuss each of the dimensions of innovation.

*Abstraction* is an important aspect of models in MDD, as we have seen in Section 4.1.1. 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 propose UML for modeling, because it does not provide the required capabilities for domain-specific modeling [25]. Instead, SF uses domain-specific languages to raise the level of abstraction. We look closer into domain-specific languages in Section 5.1.2.

*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 probably contains functionality to store and manage customer information—like a customer name, billing address, telephone number, email address, etcetera. If the granularity of components in one such software system would be low, the code that implements the functionality for customer management is likely to be scattered across numerous source files of the application. This does not allow for reuse of this code in another application, since rewriting the code may 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 [25]. 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 5.1.3.

### 5.1.2 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 [17].

> *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 [25]. 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 [22]. *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 [16] 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 now discuss the consequences of the choice for one of both types of DSLs.

Using 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 the capabilities to write the interpreter for this language are required. Parser generators can assist in the process of creating a parser, but even with assistance this is 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. With a DSL based on XML, you can make use of the XML processing functionality available in most general-purpose languages. However, this limits in the freedom of design of the language, since the tag notation used by XML is unavoidable. In both cases it is necessary to write an interpreter for the 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. The first one involves the design of 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 [17, 22]. Secondly, there are no tools 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 are 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 meant to be simple [22], 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 an executable system. When evaluated at runtime, there is no

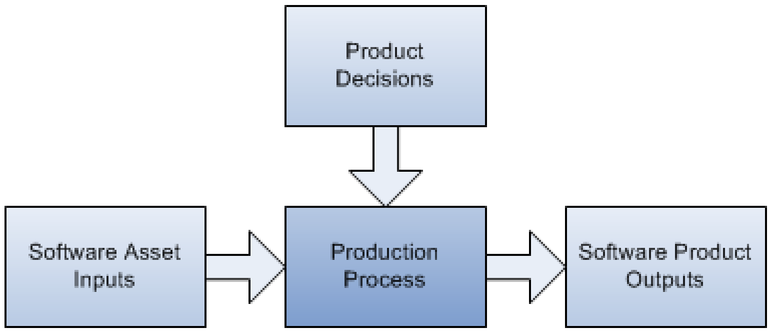


Figure 5.1: Concepts of a software product line [35]

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 a configuration parameter in the configuration file is changed, it is not necessary 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, developing an external DSL may not be a good choice, because 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, it might be worthwhile to design a custom DSL.

### 5.1.3 How Software Factories Work

Since software factories are defined to be model-driven product lines [25] 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 5.1. 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*

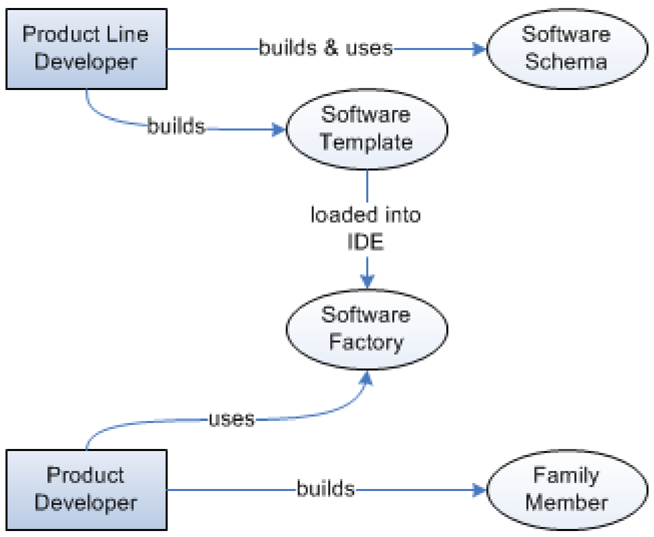


Figure 5.2: Overview of a software factory [26]

*schema*, the *software factory template*, and an *extensible IDE*. In [26] 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 used 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 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 5.2.

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 software 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 [25].

### 5.1.4 Tools for Software Factory Development

Microsoft provides *DSL Tools* [63] 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 probably needs 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.

## 5.2 Model-Driven Architecture

The Object Management Group (OMG) proposes *Model-Driven Architecture* (MDA) [40] for raising the level of abstraction at which developers design their software. Instead of DSLs, MDA uses 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 [34]. In this section we introduce the MDA process and the technologies used in this process.

This section is organized as follows. Section 5.2.1 presents the basic principles of the MDA development process. Section 5.2.2 describes how the concepts model, meta-model, and model transformation, as introduced in Chapter 4, are utilized in MDA. Section 5.2.3 introduces the technologies adopted by the OMG to support the MDA process.

### 5.2.1 Basic Principles

The MDA process does not differ much from the traditional way of software development [34]. 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 we explained in Section 4.1.1. 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, and is thus comparable with 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 [34]. 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 only have to generate a new Java-specific model 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.

### 5.2.2 Models, Meta-models and Transformations with MDA

We 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) [40]. 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 [40].

**Platform-Independent Model** A PIM describes a 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 [34]. 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 4.1.3.

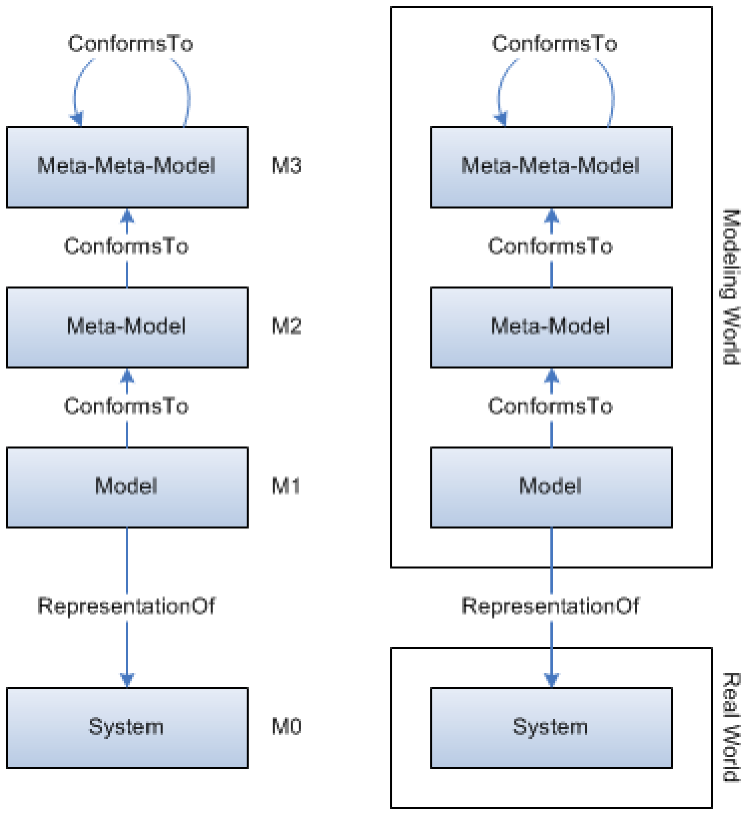
To facilitate the transformation between different types of models and between the PSM and source code, the MDA needs meta-information of the used models. Therefore MDA comes with an architecture that has several levels of meta-models. The MDA architecture consists of four layers, as illustrated in Figure 5.3(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 [5]; Figure 5.3(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 5.4 illustrates this by adding another 'modeling world', the programming language C#.

### 5.2.3 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 now give a short introduction to each of these technologies.

**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



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

Figure 5.3: The MDA Architecture

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 through 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 [24].

UML models are serializable to XMI [48], another OMG standard. XMI is an abbreviation for *XML Metadata Interchange* and is, like the name implies, based on XML [65]. These serialized files can be used as an input for transformation tools or as input for another UML tool.

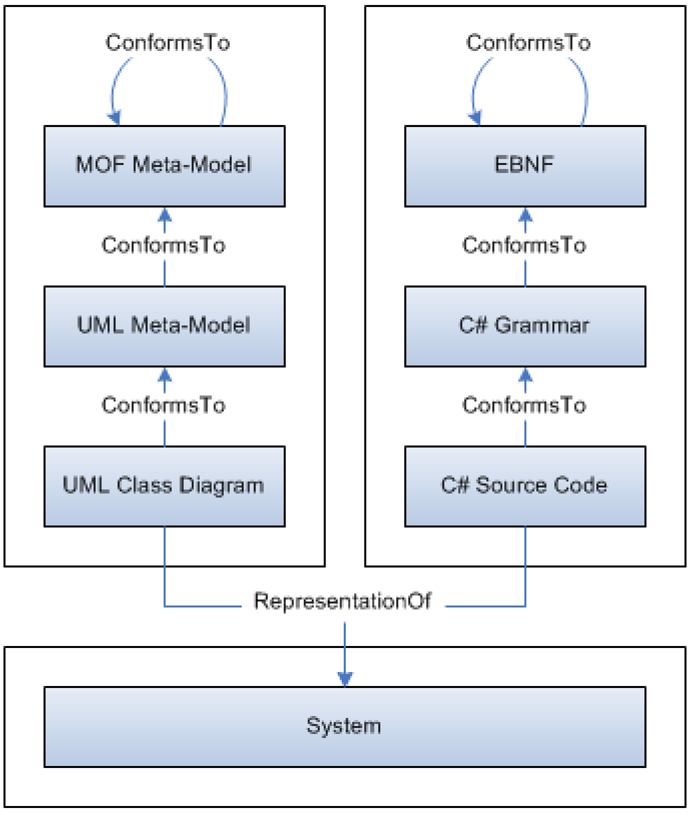


Figure 5.4: Multiple modeling worlds corresponding to a software system

**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.
- 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 [38]. 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.

**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.

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 4.1.3, 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 5.5.

**ATLAS Transformation Language** Since QVT receives quite some criticism [64], 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 5.6—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 do not further elaborate on ATL as it falls out of the scope of this document; for more informa-

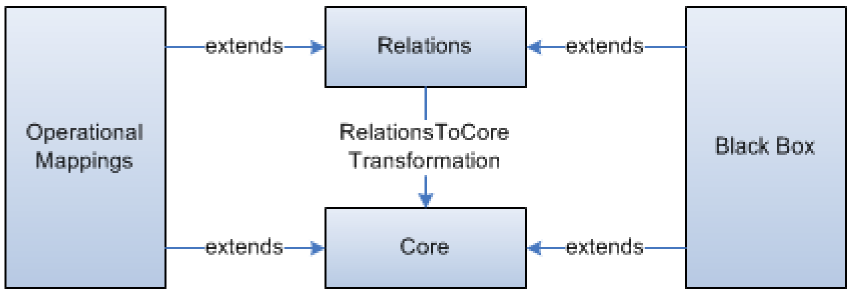


Figure 5.5: The QVT architecture

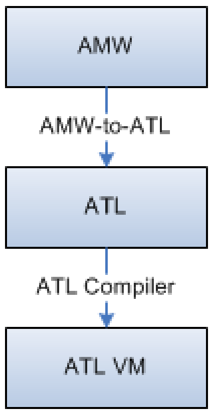


Figure 5.6: The ATL architecture

tion of both QVT and ATL, and a comparison between the two transformation languages, we refer to [32].

## 5.3 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 only present the differences in the processes, models used, etcetera. The major difference between the two is the specificity of the abstraction used in both approaches. Where MDA uses 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

| | **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 5.1: The differences between Software Factories and Model-Driven Architecture.

for a general-purpose approach allows for modeling each type of software system, but a more domain-specific approach gives a larger increase in reusability of components, and thus productivity [25]. 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 5.1 present a clear overview of the differences between Software Factories and MDA.

# Chapter 6

# Mapping Model-Driven Development on the Problem Domain

In Chapter 3 we discussed the problem domain, defined what small and simple applications are, and introduced the concepts related to service-oriented architecture. Chapter 4 presented the basic theory of model-driven development. We described two approaches in model-driven development—Software Factories and Model-Driven Architecture—and compared their main characteristics in Chapter 5. In this chapter we want to combine the knowledge of these chapters to present and evaluate the opportunities to apply MDD in our domain. At the end of this chapter, we are ready to implement one of these opportunities.

We answer three questions in this chapter. First we investigate the parts of the target application in which we can apply a model-driven solution, and decide which one of these is best (Section 6.1). Then we identify candidate MDD approaches and choose the most suitable for our project (Section 6.2). Finally we select a tool to implement the solution (Section 6.3).

## 6.1 The Choice for an Application Part

In Section 3.3 we defined what 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.* Instead of targeting one specific application layer, 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 of Endeavour applications 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; this is called model-driven service composition [50].

4. *Apply MDD in a vertical approach.* Another possibility we could investigate is a vertical approach, where we target all layers in the application. 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].

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 3.3, and are developed with the .NET version of Endeavour. Relevant metrics of Project 1, Project 2, and Project 3[3] 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.

| | |
|---|---|
| **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.

[1]http://www.rubyonrails.org/

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

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

| **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

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.

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 5.1.4—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

Now we decided to transform an initial model into a complete application, we have to decide what model-driven approach we apply to implement the generation process. We introduced the two major approaches Software Factories and Model-Driven Architecture in Chapter 5, but we are not limited to these approaches only. This section describes the choice between SF, MDA, and an alternative solution, and argues what the best approach is for this project.

With the knowledge that we have to model a complete application, we think that Software Factories is not a good solution for our problem. Because a DSL is designed to target a specific domain instead of providing a general-purpose approach in modeling software systems, and our target application is not contained in such a domain, there is probably a better solution than applying a DSL in our project.

We thought of categorizing our applications in the data-driven application domain, which should contain all applications that manage data in an underlying database. For this domain we can write a DSL to model such an application, but what would this language look like? The language should probably contain artifacts like objects, references, and tables to model the data and objects in the system. However, we already have two powerful general-purpose modeling options available, namely the UML and Entity-Relationship diagrams [12]. Choosing one of these options seems a better solution than developing a third, almost identical alternative. Moreover, designing a DSL is a time-consuming and difficult job that requires expert domain knowledge [15], and we do not think that we have enough time to gather this knowledge and implement a DSL that covers all layers of an Endeavour application.

Excluding Software Factories leaves us with the choice between MDA and an alternative solution. One such alternative is already mentioned above. Instead of using UML, we could of course take a database schema as input model and generate the target application from the tables and columns available in this model. In an extensive search for tools we found out that there are several tools available to implement a solution starting with a database. Given the data-driven nature of the applications in our domain, we were triggered to find out which of the two approaches—with UML or a database schema—is best in this project.

In the following two sections we identify the pros and cons of the MDA solution—with UML—and the approach with the database schema as initial model in our project.

### 6.2.1 Generate from a UML Model

We can choose to precisely follow the MDA development process, as discussed in Section 5.2, 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 [10]. 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 target data-driven applications in this project, we should at some point create a database. If we take a UML diagram as initial model, we have two options for implementing a database: let the developer create it manually or generate it from the UML diagram. The former approach means more work for the developer and can result in issues in the object-relational mapping (ORM), as we should generate code to store and retrieve objects in the manually created database. The latter requires us to specify transformations that take artifacts from the UML diagram as input, and in the end deliver a complete database. Since [41] proposes to design databases with UML and [4] states that the transformation from UML model to database can be automated, it should be possible to generate a database from a UML diagram. However, we expect quite a few issues in this process, because the UML diagram provides more types of artifacts and relations between artifacts than a database model does. Another issue concerning database generation is a more practical problem. If, at some point, the database changes during regeneration, we cannot afford to loose any of the data during the process. We should find a solution to this problem if we decide to generate the database from a UML diagram.

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 [28]. 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*[4] or *MyGeneration*[5] to transform the XMI exports—XML representations of the models—of a regular UML modeling tool into source code. We already searched for existing templates that transform XMI to C# source code, with little result.

We 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 |
| (–) Database and ORM might raise a few issues |
| (–) 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. 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 4.

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

[4]http://www.codesmithtools.com/
[5]http://www.mygenerationsoftware.com/

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*[6] for CodeSmith. We can use one of these existing sets of templates or choose to develop our own templates that generate source code on top of the building blocks 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 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 [10].

We 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 |

## 6.2.3 Conclusion

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

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

```
<%
string message = "Hello World!";
%>
A simple message: <%=message%>
```

Listing 6.1: A ‘Hello World!’ template

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.

## 6.3 The Choice for a Transformation Tool

With the choice for a database as initial model, we should determine how to transform this model into source code. In the previous section we already mentioned two candidates: CodeSmith and MyGeneration. Both tools are template-based code generators that provide a generic interface to databases of many database systems.

We did a few test cases with both tools and noticed that the features offered by the tools are almost identical. There are even tools available that convert CodeSmith templates into MyGeneration templates. The template editor for CodeSmith is a bit more advanced as it provides statement completion and debugging features. MyGeneration on the other hand is free—prices for CodeSmith are $99 for the Standard Edition and $399 for the Professional Edition—and already used in Endeavour for generation of stored procedures. Since MyGeneration is shipped with Endeavour and provides the same features as CodeSmith for free, we decided to deal with a less-advanced template editor and chose MyGeneration to implement our tool.

Now let us take a closer look at the features of MyGeneration. MyGeneration supports, among others, the following databases: Microsoft SQL Server, Microsoft Access, Oracle, MySQL, and PostgreSQL. The templates for MyGeneration can be written in C#, VB.NET, JScript, and VBScript. A template is a file that contains both executable code and plain text parts. To indicate that a part of text is executable code, it is placed between the tags ‘`<%`’ and ‘`%>`’. On running the template the code between these tags is executed; the code outside these tags is simply copied to the output stream. Listing 6.1 illustrates the use of templates. After execution of this piece of template code, the string “A simple message: Hello World!” is written to the output stream.

MyMeta is the interface that provides access to the schema of a database from within a template. After specification of a connection string to an instance of one of the supported

```
<%
// MyMeta is a property directly available in each template.
foreach(ITable table in MyMeta.Databases["Northwind"].Tables)
{
  %>
  Found a table: <%=table.Name%>.
  <%
}
%>
```

Listing 6.2: Template code to display the table names in the Northwind database.

```
Found a table: Category.
Found a table: Customer.
Found a table: CustomerDemographic.
Found a table: Employee.
Found a table: Order.
Found a table: OrderDetail.
Found a table: Product.
Found a table: Region.
Found a table: Shipper.
Found a table: Supplier.
Found a table: Territory.
```

Listing 6.3: Result of executing the template code in Listing 6.2.

databases, it is possible to obtain the tables, columns, foreign keys, and constraints in this database. An example of how to print all the names of the tables of the SQL Server Northwind example database with a C# template in MyGeneration is given in Listing 6.2. The result of executing this piece of template code is presented in Listing 6.3.

In the implementation phase we use MyGeneration to implement code generation from an initially created database. After the generation process we should have a completely working prototype of an application.

# Chapter 7

# Solution Criteria

To be able to perform a validation phase, we need a list of criteria that a model-driven solution in our domain should satisfy. This chapter describes these criteria and argues why they are important in our research. The list of criteria we identified is given below.

- Increase productivity
- Ensure flexibility
- Ensure maintainability
- Ensure scalability
- Target small and simple applications
- Applicable in a .NET environment

The last two items of the above list are straightforward and do not require extensive elaboration. From the central research question it is clear that we target small and simple applications, thus our solution should focus on this kind of application. Since we apply our solution to the .NET variant of Endeavour, it is also logical that our solution should be applicable in a .NET environment.

The other four criteria may need more explanation. Therefore we elaborate on the productivity, flexibility, maintainability, and scalability criteria in the following four sections, respectively.

## 7.1 Productivity

In the current development of software systems within Info Support all code is written manually. With our solution we aim to generate a large part of this manually written code from an initially created database. Since the amount of handwritten lines of code is reduced, we can expect an increase in productivity. We almost tend to say: the more lines of code we can generate, the higher the increase in productivity.

However, we should not forget that we need to design a model of the software system, before we can generate the source code for the target application. To measure a productivity increase, we should also take into account the time needed to define this model. In the worst case, the time to create the initial model takes more time than writing the generated source code by hand. Therefore we should be careful in drawing conclusions on productivity increase in the validation phase.

## 7.2 Flexibility

With flexibility we mean the degree to which the developer can alter the generated application to satisfy the requirements of the final application. In other words, the developer should be able to extend the generated source code with handwritten code to implement the functionality that is not yet available in the generated application. This requirement raises a few questions that have to be answered. Where does the developer add this additional code? What happens when he decides to run the code generator again? And, how can a developer override generated functionality? We keep these questions in mind during the implementation of the code generation tool.

## 7.3 Maintainability

Maintenance of a software application involves fixing bugs and adding or changing small pieces of code when the application is already in production. This is often an ongoing activity during the lifetime of the application. Since generated applications also require maintenance, we should ensure that these applications are properly maintainable.

Maintainability is further divided in modifiability, testability, and understandability [9]. In the validation phase we investigate the effects of our solution on each of these three characteristics.

## 7.4 Scalability

It is common that a currently small application is expanded in the future. For this reason we added the criterion that our generated applications should be scalable to an enterprise-size service-oriented application.

We want to emphasize that the changes to the system in this case are too large to be categorized under the modifiability characteristic mentioned in the previous section about maintainability. With scaling to enterprise-size we mean adding large pieces of code to implement new features for the application. These large pieces of code are likely to be new services, given the architecture of small Endeavour applications and the service-oriented focus of the Endeavour software factory.

# Chapter 8

# The Code Generation Tool

This chapter presents the tool we implemented to prove the use of a model-driven development process in our domain. We do not give a full description of the implementation of the code generator, but rather a high-level overview with a closer look on a few important aspects.

The code generation process is illustrated in Figure 8.1. Starting with two models as input—the schema of a database and a meta-data file—the tool generates a prototype of an ASP.NET web application. In the remainder of this chapter we first define the database schema and meta-data input models in Section 8.1. Then, Section 8.2 gives a short description of the target application. Section 8.3 describes how we dealt with the criteria we specified in the previous chapter. Finally, we list the limitations of our code generator in Section 8.4.

## 8.1 Input Models

As shown in Figure 8.1, the code generation tool needs two input models to be able to generate the target application. In this section we define what these models look like and argue why we need the information provided by these models. We first discuss the database schema, then we introduce the meta-data model the developer has to define in an XML file.

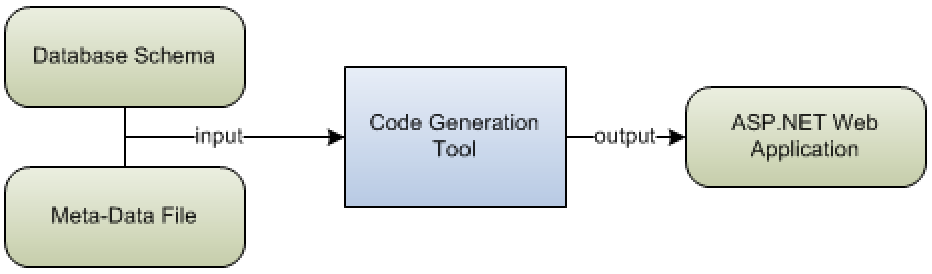


Figure 8.1: The code generation process.

### 8.1.1 Database Schema

The database schema is the main source of information for the code generation tool. With the use of MyGeneration's MyMeta interface, the tool reads the tables, columns, and foreign keys present in the database and generates source code for the target application with this information.

Because we are not aiming to develop a code generation tool that is ready to be used in a production environment, we added a few restrictions to the database schema to simplify implementation of the tool. The biggest restrictions we had to add are:

- Only single-column primary keys of type int are allowed
- Foreign keys do not refer to primary keys of the table in which they are defined

These restrictions are mainly applied to focus on the goals of our project instead of dealing with all kind of exceptional cases. We believe that with some time and effort it is possible to alter the code generator to allow databases with any kind of schema.

### 8.1.2 Meta-data

Although the knowledge available in the database schema seemed to contain enough knowledge at firsthand, we learned through a few test cases that more information is needed. In this section we define the additional input data that the developer has to specify in an XML file and explain why we need this extra information to generate our target application.

Beside a few general configuration attributes, the data in the XML meta-data file can be divided in three categories: entities and reference data, entity views, and one-to-many relations in the user interface. We elaborate on each of these data sections.

**Entities and reference objects** In the XML meta-data file the developer marks tables in the database to be represented by an *entity* or a *reference object* in the generated source code. We first explain why a developer wants different representations for the tables in the database and define the entity and reference object terms. Then we describe how the actual marking is done in the meta-data file.

The data in the database of a business application can be divided in two categories. The first category is the data that is of prime importance for the organization and forms the core of the application. It is likely that this data is changed very often, probably multiple times a day. Examples of this kind of data are customer, order, and product data.

Less important is the second type of data, which we call *reference data* [29]. This data is only in the database for implementation-specific reasons and is, for instance, used to fill list boxes or drop down boxes in the user interface. A good example of reference data is a table containing customer types. This table probably contains a few records like 'Normal Customer', 'Gold Customer', and 'Bad Payer'. Changes to this data are unlikely to occur, as it is uncommon that a new type of customer is introduced often. The sole purpose of this data is to be retrieved and put in a drop down box, so an administrator of the system can choose a specific customer type for a customer entity.

```
<entity name="Customer" plural="Customers" table="customers">
  <referenceData>
    <ref name="CustomerType" plural="CustomerTypes"
         table="customer_types"/>
  </referenceData>
</entity>
```

Listing 8.1: Specification of a customer entity with a customer type reference object.

```
<entity name="Customer" plural="Customers" table="customers">
  <views>
    <view name="list">
      <field col="Name"/>
      <field col="Address"/>
      <field col="CustomerType"/>
    </view>
  </views>
</entity>
```

Listing 8.2: Specification of a list view on a customer entity

The descriptions of both types of data imply that the source code of a class that deals with the first category of data differs from a class that works with reference data. In our project the classes representing data from the first category are called entities; the reference data is represented by reference objects. In the source code the reference objects belong to a certain entity, thus classes that work with entities should also deal with their reference objects. For instance, a class to read an entity from the database, is also responsible for reading the reference data belonging to that entity from the database.

Listing 8.1 shows how a developer defines that a table is represented by an entity, and how another table is represented as a reference object of that entity. In this case, the `customers` table is represented by the entity `Customer`, which has a reference object `CustomerType` that represents the data in the `customer_types` table.

**Entity views** To present a clear overview of entities, it is common that the user interface of an application does not show all attributes of an entity in a list view. The user has to select one row from the list to see the full representation of the entity. Because the code generator cannot decide what attributes to show in a list view, we decided to let the developer specify views on entities in the meta-data file. In the current version of the code generator the developer can only define a list view.

Listing 8.2 shows a part of the XML meta-data file that contains the list view specification on a customer entity. Imagine that the customer entity has numerous attributes, but in an overview of all customers in the system we only want to see the name, address, and customer type properties. The code example shows how to define this view.

```
<relations>
  <rel one="Customer" many="Order"/>
</relations>
```

Listing 8.3: Specification of a one-to-many relation between customer and order

**One-to-many relations in the user interface** User interfaces often have views that show a one-to-many relation. It displays the attributes of one entity, which has several sub entities that belong to that entity. In a view that displays the information of a customer in the system, for instance, it is desirable that it also shows orders that are placed by this customer.

To let the code generator write code for this kind of views, the developer has to specify this relation in the XML meta-data file. Listing 8.3 illustrates how to define this for a one-to-many relation between customer and order.

## 8.2 Target Application

The output of the generation process is a web application containing functionality to manage the data in the underlying database. The generated application has the architecture as described in Section 3.3: two cooperating loosely-coupled components, the backend and the frontend.

The generated frontend is implemented as a ASP.NET web application. This is just an arbitrary decision, the frontend could as well have been implemented with Windows Forms technology.

The backend and frontend communicate through web service technology. The choice for a web service is based on the fact that Endeavour uses this technology to implement service-oriented architecture, and we aim to generate scalable applications. We elaborate on our efforts to generate applications with proper scalability in Section 8.3.4.

## 8.3 Focus on the Criteria

We implemented the code generator with the criteria defined in Chapter 7 in mind. In this section we explain how we tried to achieve a good result for each of those criteria. Validation of our tool will prove if our model-driven solution to develop small Endeavour applications really performs well against the specified criteria.

The following four sections describe how we focused on productivity, flexibility, maintainability, and scalability during implementation of our code generator, respectively.

### 8.3.1 Productivity

Since the main goal of this project is to increase productivity, we aimed to achieve the best results possible for this criterion. From the start of the implementation phase we had the intention to generate a completely working prototype of an application. Without writing a single line of source code, it had to be possible to manage the data in the underlying database.

The final version of the tool accomplishes this goal. It does not only generate all source code for a working prototype, but also the complete solution and project infrastructure for Visual Studio. This means that the developer defines the input models, runs the code generator, and can start working with the generated source code in Visual Studio. The only activities the developer has to perform for the prototype to be operative is adding a web reference from the frontend to the web service published by the backend and compiling both solutions—the backend and frontend have separate solutions.

### 8.3.2 Flexibility

During the implementation of the tool we payed a lot of attention to the flexibility of the target application. In this section we take a closer look on our efforts to generate flexible applications.

One important requirement to achieve a flexible solution is the possibility to add custom code to the generated application that is not lost on regeneration. In our code generator we aimed to have regions in the source code where a developer writes his custom functionality or business logic and preserve these handwritten changes when the application is regenerated.

Replacing generated code with custom code can also enhance the flexibility of the generated applications. If the developer decides that a certain generated method does not provide the desired functionality, he should be able to override this method and correctly implement a custom solution. It is clear that these changes should also be preserved when a developer runs the code generator again.

Summarized, a developer should be able to do the following.

1. Add custom code to the application source code that is not lost on regeneration.
2. Overwrite generated code with custom code that is not lost on regeneration.

Since the target application of the generator is a web application, we generate both C# and ASP.NET source code. During implementation and testing we noticed that these languages need different approaches to preserve handwritten code. Below we describe the way we achieved preservation of custom code for C# and ASP.NET code, respectively.

**Preserving custom C# source code** The first approach in separating generated C# code from handwritten modifications applies the partial classes offered by the .NET 2.0 framework. Partial classes allow to specify a class over multiple source files, which gives the opportunity to put the generated code in one partial class definition and add another partial class—in a different source file—where a developer can write his custom functionality. On regeneration the file containing the first partial class definition is overwritten, the second file remains unchanged.

This seemed to work fine, until we started to think about the second requirement mentioned earlier: allowing the developer to write his own implementation of a generated method. We illustrate this with Example 8.1.

```
public partial class CustomerManager
{
    public void Save(Customer customer)
    {
        bool isValid = Validate(customer);

        // Save customer or throw exception if isValid = false.
    }

    public bool Validate(Customer customer)
    {
        return  customer.Name != null &&
                customer.Name != string.Empty &&
                customer.Email != null &&
                customer.Email != string.Empty;
    }
}
```

Listing 8.4: A generated `CustomerManager` class with validation functionality.

**Example 8.1.** Listing 8.4 presents a partial class definition of a `CustomerManager` class that is responsible for saving only valid `Customer` entities. The partial class contains a `Validate` method that checks whether a given `Customer` entity satisfies certain business rules; in this case that the properties `Name` and `Email` are not empty. The specified `Save` method calls `Validate` to check if a `Customer` entity is valid and can be stored in the database. Imagine that this partial class is generated.

While working with the generated source code, a developer notices that the `Validate` method does not check all business rules specified in the requirements of the application. He has to add additional business rules to check if the length of the name and email address properties of the customer do not exceed 50 characters. Adding these rules in this approach is difficult.

The developer cannot implement these rules in the `Validate` method of the partial class in Listing 8.4, because his custom code will be overwritten on regeneration. Since the `Save` method explicitly calls the `Validate` method, we cannot add a method containing all business rules to a new partial class definition; we can never get the `Save` method to call this method. As there is no way to overwrite the generated `Validate` method in another partial class—C# does not allow this—we cannot add the additional business rules to the manager class.

The above example makes clear that partial classes, although they provide a good solution for separation of generated and handwritten code, do not allow us to override generated methods. Therefore we implemented a solution that makes use of inheritance, and satisfies both the code separation and overriding of generated code requirements.

We decided to use a base class that contains all generated code. The developer can specify his custom code in a sub class of the generated base class. The sub class is generated the first time the developer runs the generator and is not altered when he runs the tool again.

```
public abstract class CustomerManagerBase
{
    public virtual void Save(Customer customer)
    {
        bool isValid = Validate(customer);

        // Save customer or throw exception if isValid = false.
    }

    public virtual bool Validate(Customer customer)
    {
        return  customer.Name != null &&
                customer.Name != string.Empty &&
                customer.Email != null &&
                customer.Email != string.Empty;
    }
}
```

Listing 8.5: The generated `CustomerManagerBase` class with validation functionality.

To override a generated method, the developer adds a method with the same signature to the sub class and uses the C# `override` keyword. Example 8.2 illustrates this construction.

**Example 8.2.** For this example we continue with the problem presented in Example 8.1, however this time we apply inheritance for code separation. Listing 8.5 shows the base class `CustomerManagerBase`, which contains generated code and is overwritten each time the developer runs the code generator. Both the `Save` and `Validate` methods are marked `virtual` to allow a developer to override these methods in a sub class.

The first time a developer runs the code generator, an empty class `CustomerManager` is generated; this class is a sub class of `CustomerManagerBase`. Since this class is not modified on regeneration, this is the place where the developer adds his custom validation. He overrides the `Validate` method of the base class and specifies the additional business rules to check the length of both properties. To check the generated business rules in the base class, he uses the `base.Validate(customer)` statement. After overriding the `Validate` method, the `CustomerManager` class looks like in Listing 8.6.

There are several occasions where we used base and sub classes to preserve handwritten C# code. We did not use partial classes for this purpose, because the inheritance construction fits the flexibility requirements better.

**Preserving custom ASP.NET source code** The C# language provides us with several possibilities to deal with code separation and preservation. For ASP.NET this is a bit harder as the specification of web pages is not written in C#, but in an XML-based markup language. This implies that we cannot use the inheritance-based solution applied for C# source code separation. Therefore we have to find another approach to separate generated and custom ASP.NET code. We search for a solution that satisfies both criteria listed at the beginning of this section.

```
public class CustomerManager : CustomerManagerBase
{
    public override bool Validate(Customer customer)
    {
        bool result = base.Validate(customer);

        return  result &&
                customer.Name.Length <= 50 &&
                customer.Email.Length <= 50;
    }
}
```

Listing 8.6: The `CustomerManager` sub class with extended validation functionality.

```
<asp:GridView ID="gvCustomerList" runat="server">
  <Columns>
    <asp:BoundField HeaderText="Name" DataField="Name" />
    <asp:BoundField HeaderText="Email" DataField="Email" />
  </Columns>
</asp:GridView>
```

Listing 8.7: An ASP.NET user control to display a list of customers

The final version of our code generator uses ASP.NET pages and user controls for code separation. User controls are segments of ASP.NET pages that can be reused in many other ASP.NET pages [36]. The pages contain the handwritten code and include the generated user controls. Example 8.3 gives an impression of this approach. For a detailed explanation of how ASP.NET pages and user controls work, we recommend to consult, for instance, [36].

**Example 8.3.** This example describes how source code is organized for a list view in the user interface of the target application. The view should list all customers in the database and for each customer it should show the `Name` and `Email` properties. This functionality is implemented with a page and a user control.

Listing 8.7 shows the code for the user control. The `asp:GridView` is the server control to display a list of objects and is available out-of-the-box. For each property we define a column; in this case both columns are `asp:BoundField` controls, which simply display the value of a property in the column. We omitted the code to actually bind a list of customers to this GridView to keep the example clear.

The ASP.NET page includes the user control and allows for extension with custom code; this is presented in Listing 8.8. The `Register` tag adds a tag for the user control to the set of allowed tags. In this case we specified this tag to be `cgt:CustomerGridView` and use it to include the user control in this page. The developer can now add his custom markup to the page without the fear of losing it on regeneration, as the page source file is never overwritten.

```
<%@ Register  TagPrefix="cgt" TagName="CustomerGridView"
              Src="~/UserControls/CustomerGridView.ascx" %>

<h1>Customers</h1>
<p>Below are all the customers in known in the system.</p>

<cgt:CustomerGridView ID="CustomerGridView" runat="server"/>

<a href="~/Default.aspx">Back to home</a>
```

Listing 8.8: An ASP.NET page that uses the user control from Listing 8.7

If we look at the two requirements listed in the beginning of this section, we conclude that we partly satisfy the first requirement and do not support overriding of generated code, the second requirement, at all—apart from the undesirable case where we remove the `cgt:CustomerGridView` tag from the ASP.NET page—and thereby all generated functionality—and write the whole solution by hand.

We can add custom code to the web site, but only to the ASP.NET page. If we would manually add a new column to the GridView in Listing 8.7, it would be overwritten on regeneration. To allow handwritten code in the user controls, we made use of the preserve regions offered by MyGeneration. Preserve regions are places in source code files that will not be overwritten on regeneration. The handwritten code has to be placed between tags to let MyGeneration know that it should not overwrite it. In a MyGeneration template you define where the preserve should be generated. Example 8.4 illustrates the use of preserve regions.

**Example 8.4.** We continue with the user control code from Listing 8.7 and show how the preserve regions offered by MyGeneration are applied to manually add extra columns to the GridView. The new code for the user control is presented in Listing 8.9. Imagine that the code generator outputted this code without any code between the preserve tags `PRESERVE_BEGIN` and `PRESERVE_END`. Now a developer decides that the GridView needs an extra column and adds the new column for the property `MyProperty` between the tags. On regeneration all code in the user control is overwritten, except the new column between the preserve tags.

Preserve regions give us better results in code separation as we can add custom code on more places now. How our solutions for C# and ASP.NET code preservation perform in practice, becomes clear in the case study presented in Chapter 9.

### 8.3.3 Maintainability

In [9] the maintainability characteristic is further divided in modifiability, testability, and understandability. In this section we explain how we dealt with each of these sub characteristics during the implementation of the code generation tool.

```
<asp:GridView ID="gvCustomerList" runat="server">
  <Columns>
    <asp:BoundField HeaderText="Name" DataField="Name" />
    <asp:BoundField HeaderText="Email" DataField="Email" />
    <%--PRESERVE_BEGIN HandwrittenColumns--%>
    <asp:BoundField HeaderText="MyColumn" DataField="MyProperty" />
    <%--PRESERVE_END HandwrittenColumns--%>
  </Columns>
</asp:GridView>
```

Listing 8.9: Using MyGeneration preserve regions in the user control

**Modifiability** The modifiability characteristic describes to what extend the application is modifiable during maintenance. This is almost identical to the flexibility characteristic we presented in Section 7.2. The only difference is that flexibility is about altering source code during the development process, instead of as a maintenance activity. However, since both characteristics involve changing the source code of the system, the measures for flexibility described in Section 8.3.2 have the same effect on modifiability.

**Testability** We did not put any extra effort in achieving a proper result in testability of the generated application. More research in testability of models and generated source code is required, as we decided to focus on the other characteristics due to time constraints.

**Understandability** Finally we look at the understandability characteristic. In our measures for proper flexibility we described the need for separation of manual and generated code. We also mentioned that developers never alter generated source code, as their changes are overwritten when the application is regenerated. Developers therefore only have to know what the generated code does, and the actual implementation of the functionality can be hidden.

The code generation tool generates documentation to describe the generated source code and thereby contributes to understandability of the system. This involved generating descriptions in XML that are placed above the methods and classes they describe. A tool like NDoc[1] gathers all these XML descriptions and displays the contents in one help file that forms the complete code documentation for the application. With this file the developers can understand how the generated source code functions.

To hide the generate source code from developers we used code-behind files. In Section 8.3.2 we presented the approach taken in separating manual and generated code: a base class contains the generated code and the class that holds the handwritten code inherits this base class. The file containing the generated base class is a code-behind file of the file with the manual code.

Figure 8.2 illustrates how code-behind files are presented in the Solution Explorer of Visual Studio. The `KlantManager.cs` file has a code-behind file `KlantManagerBase.cs`

[1] http://ndoc.sourceforge.net/

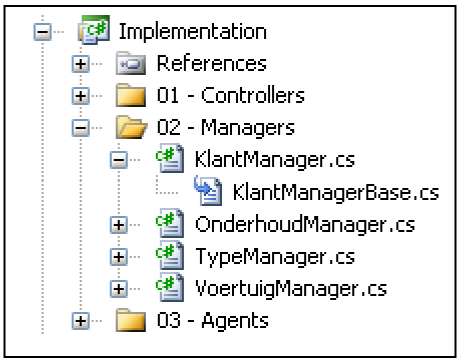


Figure 8.2: Visual Studio's Solution Explorer with four code-behind files.

containing the generated source code for the `KlantManager` class. The other three source files—`OnderhoudManager.cs`, `TypeManager.cs`, and `VoertuigManager.cs`—also have a code-behind file, but these are not visible.

With code-behind files we do not completely hide the source code from the developer, but it requires more steps to reach it. By putting the generated code behind the file that should contain the handwritten code, we hope to make clear where the developer has to write his custom code. We also added comments in both classes that explain the separation once more.

We are aware that separating code that should logically be defined in the same file, has a negative effect on the understandability of the system. Developers now have to browse through two source files to view all member implementations of a specific class, instead of just one. This is however unavoidable if we want to separate generated and manual code.

### 8.3.4 Scalability

To generate properly scalable applications, we decided to follow the design guidelines of Endeavour. This allows for expansion of the application with new services, because of the loose-coupling provided by the service-oriented architecture. Adding multiple services does require a service bus for proper registration and lookup of services in the application. Since a service bus is overkill for connecting the two services in our target application, we directly connected them with a web service and left the option open to add the bus on expansion of the system.

## 8.4 Limitations

For our research it is not necessary to deliver a tool that is ready to be used in a production environment. We only need a means to prove the concept of model-driven development for productivity increase in the software development process. Therefore we built a proof-of-concept that has several nice features, but also a fair amount of limitations. This section describes the biggest limitations.

First of all, we concentrated on a select-update cycle for data management in the generated application; it is not possible to add new entities or delete existing entities. Due to time constraints we were not able to implement all CRUD operations—create, retrieve, update, and delete—if we also wanted to provide other desirable features. Because a select-update cycle does require us to implement a fully-fledged application with validation both at the client-side and server-side, we think that our tool satisfies as proof-of-concept. Implementing insert and delete functionality in the code generator is only a matter of time and effort.

Another issue that makes our tool a proof-of-concept rather than a tool ready to be used in a production environment, is the lack of infrastructural functionality. We did not pay any attention to smart exception handling, logging, and security of the web application. Again, the limited amount of time made us concentrate on more challenging matters, like how to generate flexible applications.

Finally, the tool is limited to databases that satisfy the restrictions for database schemes, as we explained in Section 8.1.1.

# Chapter 9

# Case Study: The Garage Case

The first test for our code generation tool is Info Support's Garage Case. The Professional Development Center defined the Garage Case as a means to prove the use of state-of-the-art technologies in the Endeavour software factory. The implementation of the case is then used for demonstrating the best practices in applying the technology in Endeavour projects. We decided, in conformity with the PDC, to apply our code generator to a part of this case. Because this part consists of approximately 50 function points and does not contain difficult business logic, it is a good candidate for our case study.

In Section 9.1 we describe the process of developing the Garage Case application. The evaluation of the tool in this case study is presented in Section 9.2. Finally, we draw conclusions in Section 9.3.

## 9.1 Development Process

This section describes the development process of implementing the Garage Case web application with our code generation tool. We start by defining the two initial models, the database schema and meta-data file, and then describe the post-generation activities.

### 9.1.1 Database Schema

The first step in developing a web application with the code generator is to create the SQL Server database. We took the database model of the Garage Case and modified it to match the restrictions described in Section 8.1.1. The database schema used as input for the code generation process looks like Figure 9.1. The language used for tables and columns in the Garage Case database is Dutch; Table 9.1 contains translations of the table names and a short description of each table. For a full definition of the database model, we refer to Appendix A.

### 9.1.2 Meta-data

When the database is implemented we can define the second model, which defines meta-data for the tables and relations in the database. In Section 8.1.2 we described the three

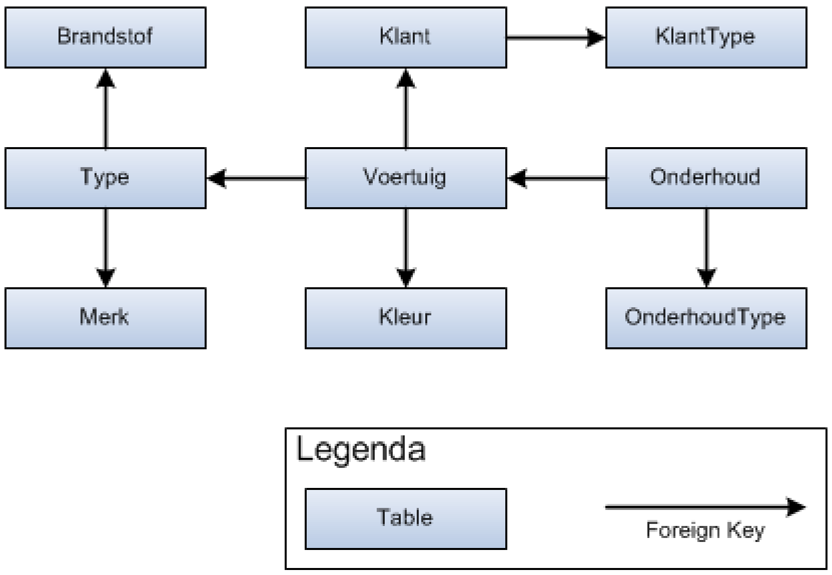


Figure 9.1: The Garage Case: Database model

| Table | Translation | Table | Translation |
|---|---|---|---|
| `Brandstof` | Fuel | `Onderhoud` | Maintenance |
| `Klant` | Customer | `OnderhoudType` | MaintenanceType |
| `KlantType` | CustomerType | `Type` | Type |
| `Kleur` | Color | `Voertuig` | Vehicle |
| `Merk` | Brand | | |

Table 9.1: Translations for Garage Case database table names.

categories of meta-data that are specified in the XML meta-data file. We explain how we dealt with each of these categories in the Garage Case.

The first step in specifying the meta-data file is identifying entities and reference objects. Figure 9.2 illustrates the entities in the Garage Case, their reference objects, and interrelations. This model is represented in the XML meta-data file and provided to the code generator at the beginning of the generation process.

Then we define a list view for each entity identified in the previous step. In Listing 9.1 we show the XML for the `Voertuig` entity, including the list view definition. The list view consists of four columns `Naam` (name), `Kilometerstand` (mileage), `Kenteken` (license number), and `Verkoopdatum` (sale date). The code listing also shows how the `Kleur` reference object is defined as reference data for the `Voertuig` entity.

Finally, we specify the one-to-many relations displayed in Figure 9.2 in the XML file. We included the full specification of the XML meta-data file in Appendix A.

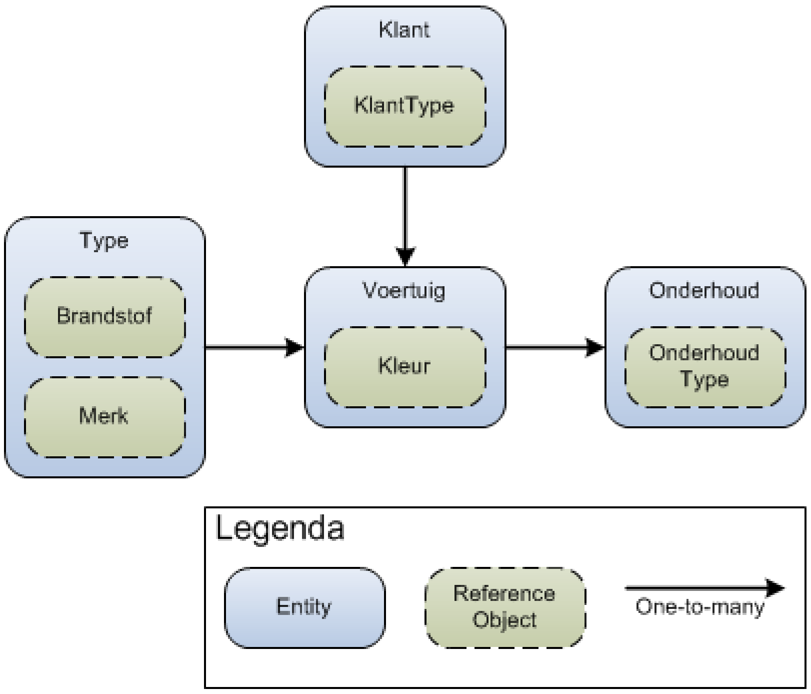


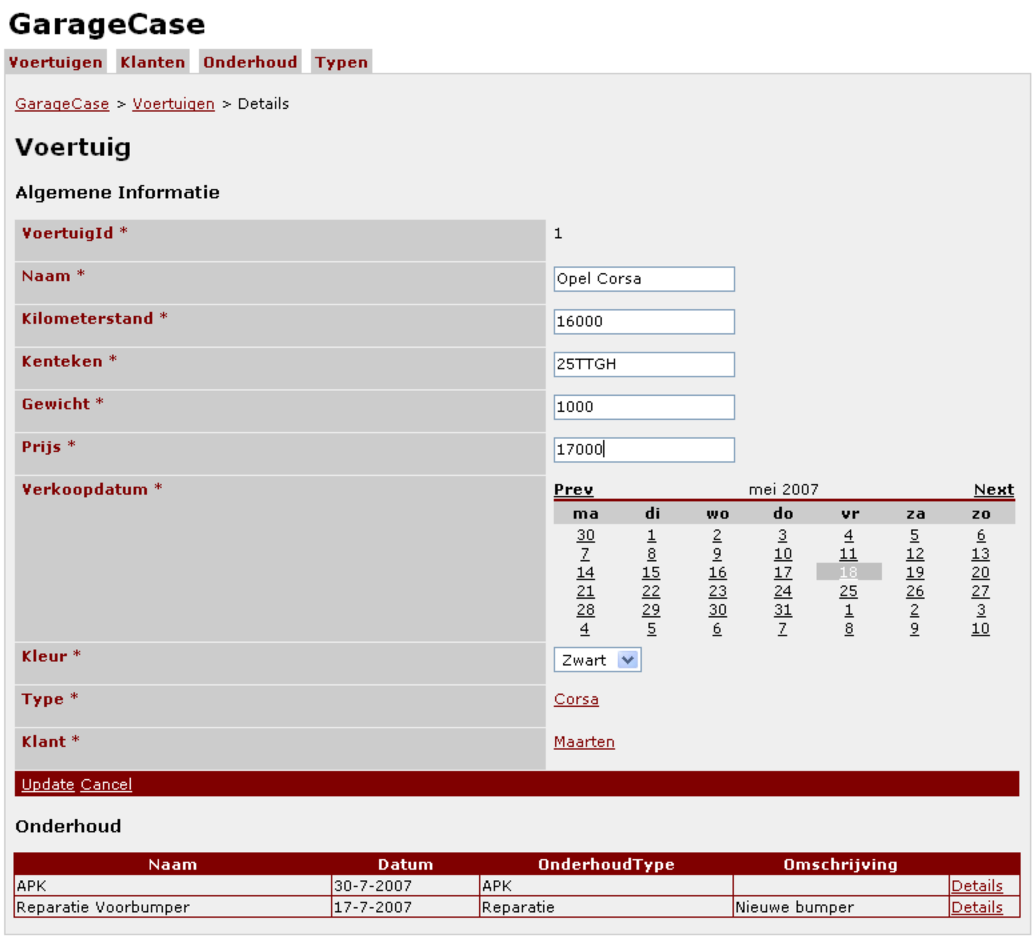
Figure 9.2: The Garage Case: Entities and reference objects

```
<entity name="Voertuig" plural="Voertuigen" table="Voertuig">
  <views>
    <view name="list">
      <field col="Naam"/>
      <field col="Kilometerstand"/>
      <field col="Kenteken"/>
      <field col="Verkoopdatum"/>
    </view>
  </views>
  <referenceData>
    <ref name="Kleur" plural="Kleuren" table="Kleur"/>
  </referenceData>
</entity>
```

Listing 9.1: Specification of entity `Voertuig` in the XML Meta-data File

### 9.1.3 Post-Generation Activities

With both input models defined, we ran the code generator and obtained the source code for the Garage Case application. After building the application and specifying a style sheet, we ended up with a working prototype that is able to manage the data in the underlying database. An impression of what this web application looks like is given in Figure 9.2,


GarageCase

Voertuigen Klanten Onderhoud Typen

GarageCase > Voertuigen > Details

Voertuig

Algemene Informatie

| | |
|---|---|
| VoertuigId * | 1 |
| Naam * | Opel Corsa |
| Kilometerstand * | 16000 |
| Kenteken * | 25TTGH |
| Gewicht * | 1000 |
| Prijs * | 17000 |
| Verkoopdatum * | Prev mei 2007 Next |
| Kleur * | Zwart |
| Type * | Corsa |
| Klant * | Maarten |

| ma | di | wo | do | vr | za | zo |
|---|---|---|---|---|---|---|
| 30 | 1 | 2 | 3 | 4 | 5 | 6 |
| 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Update Cancel

Onderhoud

| Naam | Datum | OnderhoudType | Omschrijving | |
|---|---|---|---|---|
| APK | 30-7-2007 | APK | | Details |
| Reparatie Voorbumper | 17-7-2007 | Reparatie | Nieuwe bumper | Details |

Figure 9.3: The Garage Case: Modifying a vehicle

which displays the form to alter a `Voertuig` entity.

To finish the Garage Case application, we only have to alter the generated prototype to satisfy the requirements for the final application. This means that we manually write source code to extend or change functionality of the web application. Performing these additional activities gives us insight in the flexibility of the generated applications.

Because the main goal of this case study is not to develop an application that completely satisfies the requirements of the Garage Case assignment, we identified a number of extensions to the generated source code that are likely to occur and experimented a little to find out the effort needed to implement them. We listed these extensions below.

- Add a new table to the database.
- Add a new column to a table of the database.
- Add additional business rules to validate user input.

- Change the controls used to alter data in the user interface.
- Change the graphical layout of the web site.
- Alter the object model of the application.

The first two extensions are likely to occur during the development of the application. The developer generated a prototype from an initial database and later in the process he decides that an extra table of column is needed. Adding tables and columns requires regeneration of the prototype.

The other four extensions involve adding or changing source code manually. In the evaluation of the flexibility of the generated applications we see how our tool performed for all these extensions.

## 9.2 Evaluation of the Code Generator

This case study gave us a first impression of how the code generator performed in developing small Endeavour applications. In Chapter 7 we listed a number of criteria to validate a model-driven solution: increase productivity and ensure flexibility, maintainability, and scalability. We use these criteria to evaluate the performance of the tool in this test case.

In the following four sections we describe the experiences acquired during implementation of the Garage Case application for each of the identified criteria.

### 9.2.1 Productivity

After specification of the two input models and running the code generator, we already had a working prototype of the Garage Case web application without writing a single line of source code. To complete the application we only had to specify the style of the web site in a Cascading Style Sheet (CSS) file. This suggests a high increase in productivity, but let us take a closer look before drawing conclusions to fast.

In the original development process a developer first creates a database and then writes the SQL, C#, and ASP.NET source code. Using the code generator still requires the developer to create the database manually, but instead of writing the complete source code he only has to specify meta-data in an XML file. From these models, we approximately generated 5,500 LOC (SQL, C#, and ASP.NET code), where we needed only 69 lines of XML meta-data. We should note that the generated code is a little more bloated than when it would be written manually because we needed some extra code to ensure flexibility and maintainability of the generated application, but this is probably not more than 500 lines of code. If we subtract this 500 LOC from the total 5,500 LOC that was generated, we have around 5,000 LOC that would have had to be written manually if we did not apply our model-driven approach. This conforms to the measure of 120 LOC per FP that Info Supports uses in their software size estimations, as the difference of 1,000 LOC ($120 * 50 - 5,000 = 1,000$) can be explained by the fact that we did not implement proper logging, exception management, authentication, and authorization.

Section 7.1 stated that we should be careful in drawing conclusions on productivity increase from a decrease in the lines of code that have to be written manually. However, the reduction in this case is very large—from 5,000 LOC to 69 LOC—and the meta-data definitions are not at all difficult. Thus we can say that writing the straightforward definitions in the XML meta-data file costs a lot less effort than implementing the whole application manually. We conclude that the increase in productivity *in this test case* is large.

### 9.2.2 Flexibility

Validating for flexibility means that we should check whether it is possible—and relatively easy—to add custom functionality and logic to the application after the initial generation process. In this section we present the flexibility strengths and weaknesses of our code generation approach encountered in this case study.

We now describe how our tool performed for each of the extensions we identified in Section 9.1.3.

**Add new tables and columns** When a developer decides to add a new table or column to the database after the initial generation, he has to alter the database and meta-data models and regenerate the prototype. Without losing any of his manually written source code—provided that he wrote the code in the right places—the application now has additional pages and controls to deal with the new tables and columns.

**Add additional business rules** The code generator already generates basic entity validation at the client-side and server-side of the application. It is likely however that additional business rules are needed for proper validation of user input. We illustrate this with an example where we needed to implement additional entity validation by hand.

In the Garage Case we have several integer fields that are not allowed to have values smaller than zero. Examples of these fields are the mileage, price, and weight attributes of the vehicle entity. The code generator does generate code to check if these attributes are not empty or not of type integer, but not for validating if the value is larger than zero. We had to implement this functionality by hand, which required us to alter code in both the backend and frontend.

We used the possibility to override generated functionality for extending entity validation in the backend. In the previous chapter we gave an example of how this should be implemented (Listing 8.6).

The frontend required us to add validation controls to the ASP.NET user control that allows for modifying `Voertuig` (vehicle) entities. As we mentioned earlier, it is not possible to override the ASP.NET markup code like we have done for C#. Therefore the code generator applies preserve regions for manual code preservation. Section 8.3.2 explained the use of preserve regions, but we now illustrate how preserve regions are used for allowing additional validation controls in Example 9.1.

**Example 9.1.** If we look at the `Kilometerstand` (mileage) field in Listing 9.2, we see that the code generator already generated basic validation controls for the text box: the

```
<asp:DetailsView ... >
  <Fields>
    <asp:TemplateField HeaderText="Kilometerstand">
      <ItemTemplate>
        <asp:Label ... />
      </ItemTemplate>

      <EditItemTemplate>
        <asp:TextBox ID="txtKilometerstand" ... />

        <asp:RequiredFieldValidator
            ControlToValidate="txtKilometerstand" ... />

        <asp:CompareValidator
            ControlToValidate="txtKilometerstand"
            Type="Integer"
            Operator="DataTypeCheck" ... />

        <%--
        Write custom validators for the Kilometerstand field between
        the PRESERVE tags to avoid overwriting on regeneration.
        --%>
        <%--PRESERVE_BEGIN KilometerstandValidators--%>

        <asp:RangeValidator
            ControlToValidate="txtKilometerstand"
            MinimumValue="0"
            Type="Integer" ... />

        <%--PRESERVE_END KilometerstandValidators--%>
      </EditItemTemplate>
    </asp:TemplateField>
  </Fields>
</asp:DetailsView>
```

Listing 9.2: Preserve custom validation controls in the frontend

`RequiredFieldValidator` and the `CompareValidator`. These controls check if the text box with identifier `txtKilometerstand` contains a value, and if this value is of type integer, respectively. However, it is impossible for vehicles to have a negative mileage, so users should not be allowed to enter integers smaller than zero. Therefore we specify an additional validation control—a `RangeValidator` with its minimum value set to zero—between the preserve tags, to implement this business rule. On regeneration the new validator will not be overwritten.

We conclude that specifying additional business rules for entity validation in the Garage Case is relatively easy and intuitive.

**Change graphical layout of the web site** It is likely that the final application has to look different than the plain, white design of the web site after generation. Because the code generator structures the output HTML of the ASP.NET pages and controls in such a way that all visual style definitions are specified in a separate CSS file, the developer has complete freedom in designing the web site. He can even use the page designer available in Visual Studio to design the web site in a WYSIWYG—What You See Is What You Get—approach.

Apart from using CSS style definitions, the developer can also add his own HTML to the ASP.NET pages to customize the looks of the web application. Because the ASP.NET pages are never overwritten, the developer does not lose his changes on regeneration of the application.

Since the tool generates pages and controls based on the tables that are available in the database and the specified meta-data, it is not possible to completely restructure the web application and still support regeneration of the application.

**Change edit controls in the user interface** In the generation process the code generator decides what user interface control—a text box, a list box, or a drop down box, for example—is used for each column type in the database. For instance, a text box control is used for columns of type `varchar` and columns of type `datetime` are altered with an ASP.NET Calender control.

If a developer wants to change the control used for altering data—like using three text boxes for modifying date fields, instead of the ASP.NET Calender—he currently has a hard time. These controls are defined in the ASP.NET markup language, and thus we need MyGeneration tags to preserve the handwritten changes. However, using large numbers of preserve regions in a single file, results in confusing source code that is hard to maintain. For this reason, we decided to search for another solution.

In the end we did not implement any support for preserved changes of these controls, as we did not have enough time to design and implement a solution. We think that the best solution is to let the developer specify a control for each column in the database; these definitions should be included in the meta-data file. This way he has full control over the used controls for altering data in the user interface.

**Alter the object model of the application** The last extension we discuss involves the object model of the generated application. In the domain analysis in Chapter 3 we stated that we are allowed to assume that the object model of the application is identical to the schema of the database. This means that a table for customers in the database is transformed in a customer entity in the object model, for example. There are however multiple occasions where it is more logical that these two models are not identical. The limitation of not being able to let these two models differ, definitely has a negative effect on flexibility.

### 9.2.3 Maintainability

In this section we evaluate the maintainability of the target application using the sub characteristics modifiability, testability, and understandability.

For the modifiability sub characteristic of maintainability we refer to the flexibility evaluation above, because the same findings hold for modifiability as we explained in Section 8.3.3.

We can say something about testability of the application source code, although we did not focus on this characteristic during the implementation of the tool. Both the generated and handwritten methods are testable with ordinary unit tests, as the implementation of both kinds of methods are visible to the developer. For writing a test method for a generated method the developer has to interpret generated code, which can be very hard if the generated code is badly structured and thus unreadable. We aimed to generate clean code as much as possible and expect that it is readable enough for test developers to write their unit tests. We validate this goal in the experts' opinion, presented in the next chapter.

If we apply the metrics identified by Bruntink and van Deursen [11], we get a better view on the testability of our generated systems. The size-related metrics for a class—number of lines of code, number of fields, and number of methods—correlate with the test-case metrics; larger classes with more fields and methods, result in larger and more difficult test cases. Since the structure and architecture of the generated source code conforms to the principles of Endeavour, these metrics do not differ compared with a handwritten solution.

For the inheritance-based metrics, there is one difference between our generated and the traditionally handwritten solutions. We had to apply inheritance to ensure code preservation in the business service, as explained in Section 8.3.2. This can slightly increase test case size, but we expect this increase to be small, because it involves only one new subclass for a limited amount of classes.

The metrics for external dependencies and class quality do not differ from the traditional approach, and therefore do not indicate larger or more difficult test cases. In the end we do not expect a high decrease of testability, compared with the handwritten Endeavour applications.

Finally, we should be careful in saying that the source code of our generated applications is properly understandable, because we have been working with this code for several months and are familiar with its structure and style. Therefore we leave the understandability evaluation to experts that have not seen any of the generated code yet. In the next chapter we describe the results of this evaluation.

### 9.2.4 Scalability

Since we did not have time available to expand the Garage Case application to enterprise-size, we cannot say anything about scalability based on this case study. We validate for scalability in the next chapter, where we ask the opinions of several domain experts.

## 9.3 Conclusions

Now we weigh up the positive and negative aspects of our tool in this case study, and give a conclusion for its overall performance. First we summarize and rate the results for each of the criteria; the scores range from 1 (very bad) to 5 (very good).

**Productivity** The generation process greatly reduced the amount of manual lines of code we had to write, and therefore significantly increased productivity. Due to a decreased flexibility, we needed more time than usual to add the features that were not generated. In the end, the overall productivity increase was good. Score: **4**.

**Flexibility** Flexibility is the biggest issue in this case study, as it will probably be in other cases as well. It is quite hard to add certain functionality manually, especially in the frontend. There are even features that cannot be implemented without losing the possibility to regenerate the code from the models. Score: **2**.

**Maintainability** Influenced by the decreased flexibility, the maintainability of the application also deteriorated. We do think that the generated source code is properly understandable and testable. Score: **3**.

**Scalability** Although we do not expect serious issues in expanding the application, we cannot score for this criterion because we did not do actual expansion in this case study. Score: **?**.

Overall we think that the tool would have performed a lot better if we could achieve a higher flexibility of the generated applications. Higher flexibility means less effort to implement custom functionality by hand, and thus higher productivity. Since the modifiability sub characteristic of maintainability is almost identical to our flexibility criterion, the maintainability of the application also benefits from increased flexibility.

There are two opportunities that, in our opinion, have a positive effect on the flexibility attribute. First, the preservation of handwritten code can be improved to allow developers to customize larger parts of the generated application, without losing these changes on regeneration. This especially holds for the ASP.NET code, as we have seen in Section 8.3.2 and Section 9.2.2 that there are quite a few issues regarding code preservation.

Second, allowing developers to specify more information in the XML meta-data file can lead to increased flexibility. With the current version of the code generator it is, for instance, not possible to define what user interface control should be used to alter the value of a column in the database. The tool has fixed relations between controls and data types of columns, thus a column of type `varchar` will always result in generation of a text box control in the user interface. Allowing a developer to specify these relations in the meta-data file for each individual column, would definitely increase flexibility of the frontend. It is likely that there are more occasions where better flexibility could be achieved by letting the developer specify more meta-data, but this requires additional research.

These are our findings based on the presented case study. In the following chapter we ask the opinion of several domain experts to get a better view on the performance of our tool. After combining the results of the case study and the experts' opinion, we elaborate more on tool performance and future work in Chapter 11.

# Chapter 10

# Experts' Opinion

To perform a proper validation of our solution, a case study alone is not enough. It is important that someone from outside the project, with knowledge in the domain, gives his objective opinion on the approach. Therefore we organized a session for four members of the PDC to receive comments on our work. In their daily activities, the participants research and implement new features for the Endeavour software factory, investigate the latest technologies, and contribute to software projects in the role of architect.

During this session we introduced the project, presented our solution, and initiated a discussion. At the end we handed out a questionnaire and asked them to write down their remarks and suggestions for our tool. This chapter describes the ideas that came up during the discussion, and were found in the answers to the questions in the questionnaire. We included the questionnaire in Appendix B.

We asked the participants to give their opinion on the approach we took in this project, and presented the results in Section 10.1. Section 10.2 describes the problems and suggestions for each of the criteria we identified in Chapter 7. Finally, Section 10.3 draws conclusions.

## 10.1 Opinion on the Approach

Before we concentrate on the suggestions for the code generator, we present the opinion of the participants on the approach taken in this project. After the research phase we had to decide what model-driven approach we were going to apply in the implementation phase, so now we are curious if the experts think that we took the right decision—using a database schema and some meta-data as input for the generation process—or that a different approach might have gained better results.

The remainder of this section contains the advantages and disadvantages of our approach, in the opinion of the members of the PDC.

Because a data model is always needed to implement a small data-driven application, our solution is a quick start in the development process. When this model and the meta-data are defined, a complete working prototype is generated and the developer only has to customize its code to make it satisfy the application's requirements. The possibility to

| | Person 1 | Person 2 | Person 3 | Person 4 | Average |
|---|---|---|---|---|---|
| Productivity | 4.0 | 3.0 | 4.0 | 2.0 | 3.250 |
| Flexibility | 3.5 | 3.0 | 3.0 | 3.0 | 3.125 |
| Maintainability | 4.0 | 3.0 | 3.0 | 3.0 | 3.250 |
| Scalability | 3.0 | 3.0 | 4.0 | 3.0 | 3.250 |

Table 10.1: Scores awarded by the participants for each of the criteria

change the data model during the development process and then regenerate the source code, is another advantage.

A disadvantage of the approach taken in our project is the tight coupling between the data model and the object model. If the data model gets more complex than, for instance, the one used in the case study, we cannot directly link tables to entities anymore. Two examples of cases where our solution will fail, are `n:m` relations and inheritance. Although we were allowed to assume in this project that these two models were identical, we agreed with the experts that more flexibility is needed if we want to apply our tool in a production environment.

## 10.2 Evaluation of the Code Generator

Like we did in our case study in Chapter 9, we asked the participants to rate our tool based on the criteria identified in Chapter 7; the results are presented in Table 10.1. We take these scores into account when drawing conclusions in the last section of this chapter.

In the following four sections we describe the opinions on how our code generator performed with respect to productivity, flexibility, maintainability, and scalability, respectively.

### 10.2.1 Productivity

The participants concluded that our solution increases productivity in developing small Endeavour applications. Programming the generated web application by hand would take a lot more time than creating the models and generating it. Generally, this type of application contains a lot of plumbing code, which is now taken care of by the tool. The generation of the solution and project structure for Visual Studio was a good decision, as this saves the developer time in organizing the generated source code.

Because the time required to develop an application with our tool is determined by the amount of handwritten code that has to be written after the initial generation process, flexibility of the generated applications highly influences the overall productivity. In the following section we describe a number of flexibility issues of the target application, and explain why they negatively affect productivity.

Finally, the increase in productivity could have been larger if the extension of the generated source code would not have had to be done in such a traditional way. Currently the developer implements custom functionality by changing and adding C# code; almost all work is done manually and without the use of frameworks. Adopting frameworks, or other mechanisms to speed-up implementing alternative functionality, is expected to benefit the productivity increase.

### 10.2.2 Flexibility

The degree to which our solution satisfies the flexibility criterion is, in the opinion of the participants, different for the business service and the frontend. The flexibility of the business service is considered good, as the developer has the choice to extend, or completely override, the generated source code. This allows him to implement his custom business rules and data access code, if needed.

For the frontend however, there is an issue. The developer has full freedom in designing a page—because of the separation of generated and custom code in pages and user controls, respectively—but there are many cases where he wants to alter the user controls as well. Currently, most of the changes a developer makes to a user control are overwritten on regeneration. To serve as a tool that is ready for production work, a developer should be able to choose what control is used for which field—for instance, using three drop down boxes for date selection, instead of the currently generated ASP.NET calendar control—and to use custom controls, like a zip code control that automatically validates the input.

A flexibility issue that influences the whole application, is the tight coupling between the database model and object model. The direct mapping from database tables to objects and user interface screens results in less flexibility for the developer to implement his custom functionality. Less flexibility means more effort to incorporate the custom code, and thus negatively affects productivity. Moreover, the direct mapping fails when we deal with constructions like `n:m` relations and inheritance. To solve these issues, the experts suggested to investigate a combination with an ORM framework. Such a framework allows the developer to specify the mapping from database tables to objects in separate files, and thereby removes the tight coupling between database model and object model. It would be interesting to research generation from these mapping files, as this offers the developer full freedom in designing the object model. We propose this research project as future work in Section 11.4.

### 10.2.3 Maintainability

For maintenance activities, the developer has all code of the application available, and is therefore not dependent on our tool. Moreover, he can perform these activities in Visual Studio, an integrated development environment he is familiar—and thus more productive—with. The generated application has a well known structure, as it conforms to the prescribed Endeavour architecture. If it is necessary to change the data model, he can modify the database and regenerate the application, without losing any handwritten code. These are all advantages of the proof-of-concept.

Maintaining the frontend is more difficult, because the developer cannot overwrite or extend all generated code without losing it on regeneration. To partially solve this problem, the MyGeneration preserve tags allow some extension of ASP.NET code. However, the presence of many of these tags in one file does not contribute to the readability of the source code, and it becomes hard for a developer to find out where he has to write his custom code.

However, before we are ready to support production work we need to answer a number of maintenance related questions. How do we deal with the code generation tool? Do we maintain it alongside the application—thus maintain the templates specific for each application—or separate from the generated applications? What happens if MyGeneration is no longer supported? How do we deal with multiple versions of the meta-data file? These questions currently remain unanswered and require more research.

### 10.2.4 Scalability

A question that came up during the discussion was whether our tool would still work for databases with many tables. While developing an application it is not desirable to have a generation process that takes a long time before it is finished. Since we did not perform any tests with a large number of tables, we could not directly answer this question. After the demo presentation we decided to test our code generator's performance with a database of 50 tables. The generation process for this database took approximately three minutes. In our opinion this is a reasonable result, because a developer only has to run the process again when the database schema changes.

An issue that significantly influences the scalability of the applications developed with our code generator, is the mismatch between the data view taken in our approach and the service-oriented view used in enterprise SOA applications. Where the generated code outputted by our solution is purely focused on data management, a service-oriented application would concentrate more on business processes, as we saw in Section 3.2.1. The database schema we use as initial model is a pure static representation of the system, it does not contain any information on business processes or use cases. Scaling an application developed with our code generator requires the developers to incorporate business processes in the business logic of the application. The current generated business service is just a component providing CRUD functionality, and should be wrapped within a new business service to satisfy the service-oriented view on software. This is possible though, but it does require some effort.

## 10.3 Conclusions

From the discussions with the domain experts we conclude that, in their opinion, the project is a good start. However, to transform the proof-of-concept developed in this project into a tool that is usable in a production environment, there are quite some issues that should be solved. This also becomes clear from the scores awarded in the questionnaire; all are slightly above average.

Most of these issues are related to the flexibility of the generated applications. The tight coupling between the database model and object model, and the handwritten code preservation issues in the frontend, are the biggest flaws in the current solution. The experts proposed to investigate a combination with frameworks and extension of the meta-data definitions to solve the flexibility issues. In the future work section of Chapter 11, we discuss a number of possible future research projects in more detail.

During the discussions we also got feedback on how our tool would perform if we would apply it in developing an enterprise SOA application. Although this falls outside the scope of this project, we think it is valuable information and deserves some attention. In an enterprise SOA environment the business services are not CRUD-based, like the backend generated by our solution. However, in the lower parts of a service we still need CRUD functionality, so we can use our tool to generate a part of business services in a SOA. The service layer then has to be handwritten, or modeled in another modeling language. Generating the frontend from a database model is not desirable in a SOA, as it is common that a frontend is built on an existing service for which no database model is available.

# Chapter 11

# Conclusions and Future Work

## 11.1 Conclusions

In this project we researched the applicability of model-driven development to increase productivity of the software development process of small and simple Endeavour applications. Because this type of application is mainly data-driven, we decided to use a database model as initial model. The developer creates a database, specifies a set of meta-data definitions, and a prototype of a web application is generated. The generated source code is considered a starting point. Because the generation process also outputs solution and project files, the developer loads the source code into the Visual Studio IDE and is ready to implement additional functionality and business rules.

A case study and the opinion of the experts proved that *in our domain* this led to a significant increase in productivity. The overall productivity of developing an application with our code generator is determined by the amount of work needed to implement the functionality not provided by the initial generation. Therefore, the flexibility of the source code of the generated applications influences the overall developer productivity; the less flexible the generated applications are, the more effort it costs to implement custom functionality. We managed to generate flexible application backends, where we used an inheritance construction for handwritten code preservation. The frontend of the generated application is less flexible, as we did not achieve full separation of handwritten and generated code. Solving these flexibility issues is expected to result in a higher overall productivity.

Generating the applications in this domain does not jeopardize their scalability and maintainability, although there are a few issues that need to be fixed. For better maintainability we should find a solution for the code preservation issues in the frontend of generated applications, because complete preservation of handwritten changes to the source code is not yet supported. Performing the maintenance activities manually is possible, as the generated code is properly readable and structured. Scaling the applications to an enterprise SOA level requires the developer to wrap the currently generated backend within a service providing more service-oriented functions, because the CRUD-based approach does not fit the service-oriented view on software. This view prescribes that services should offer functions related to business processes, not to actions on the database.

| Good | Bad |
|---|---|
| Quick generation of source code, that serves as starting point<br>Generation of application structure and project files<br>Regeneration when model changed | Traditional way of implementing custom functionality<br>Tight coupling database schema and object model |

Table 11.1: Good and bad characteristics of our solution.

To remove all kind of exceptional cases and focus purely on the concept of generation from a database model, the domain targeted by our tool is heavily scoped. Because we had limited time for the implementation, the tool does not generate create and delete functionality for the target application, only retrieving and updating of business entities is supported. This way we had time left to look into other interesting issues, like generating basic entity validation from column properties in the database. We also applied a number of restrictions on the input database schema. The current version of our tool will, for instance, fail for a database with multiple columns as primary key, or with foreign keys that reference the table they are contained in. Finally, we did not invest time in generating proper caching, authorization, authentication, and logging for the target applications.

During discussions with the domain experts, we learned that our tool is not ready for production work, even if we fixed all the issues mentioned above. The restrictions on the input database schema limit the use of the generator in practice. However, if we remove these restrictions we cannot use the direct mapping from database tables to objects anymore, because there are constructions that are often used in object models that are modeled differently in the database. Examples of such constructions are `n:m` relations and inheritance. To support development in a production environment, we should remove the tight coupling between database schema and object model.

We summarized the good and bad points of our solution, with the adaption to a broader scope in mind, in Table 11.1. In the future work section we propose research projects to address the presented bad points of our approach.

With these notions in mind, we answer the central research question defined in the beginning of this project. We copied this question below.

*How can we apply model-driven development to increase productivity in developing small and simple Endeavour applications without losing the quality level achieved with the original approach?*

We can conclude that using a database schema as initial model, together with a set of meta-data definitions, results in a significant productivity increase in the defined domain, without deteriorating the quality level of the generated applications. However, we also learned that the domain is too heavily scoped to compare it with a production environment. Further research is required to broaden the scope and work toward a tool that is ready for more advanced scenarios.

## 11.2 Contributions

For Info Support we investigated one possible direction in their search for productivity increase in small application development. We delivered a proof-of-concept that shows the power of code generation from a database schema, but also a large number of issues that have to be solved. Although we did not deliver a tool that is ready for production work, we obtained a lot of valuable knowledge and ideas for future projects. Because Info Support did not have any experience in generated complete applications, this project is a first step in the right direction. A number of future research opportunities are discussed in Section 11.4.

## 11.3 Reflection

At the end of a project it is common to look back and evaluate the overall process. This way we can learn from the mistakes made in this project, and do things differently in the future. We first discuss the things that went good, and then describe the points that were not satisfactory.

Since this the first time that we performed a research project of this size, we think that we did quite well. We performed a thorough literature study, and tried to apply the gathered knowledge in the remainder of the project. The proof-of-concept delivered with this project is, even with an upgrade, not ready for production work, but a good basis for future research.

Of course, there were several things that we could have done better. Firstly, we think that we should have invested more time in comparing the approaches identified in the analysis phase. A number of test cases with Microsoft's DSL Tools and an MDA tool could have improved our view on the possibilities of Software Factories and MDA for our project.

Secondly, we would have liked to organize an empirical study to validate our solution, instead of the applied experts' opinion. With a few cases implemented by Endeavour developers, we think that we would have had a more complete list of issues of our tool and the approach taken. However, we are aware that we did not have the time and resources to organize such a study, and that the experts' opinion was the best alternative in this project.

Finally, we learned quite early in the project that code generation on top of frameworks is an alternative worth looking into. Unfortunately, in the beginning of the project the idea was to compare our MDD solution with the solution resulting from Bastiaan's study about frameworks, and applying both frameworks and MDD for our proof-of-concept implied an unfair comparison. In the end we abandoned this plan because both solutions were not comparable anymore—where we aim to generate a full working prototype from an initial model, Bastiaan's framework approach targets specific spots in an application—but we already made the decision not to apply frameworks.

## 11.4 Future Work

For Info Support we propose research in two directions. Firstly, the approach taken in this project—generate small and simple applications from an initially designed model—needs more research. Table 11.1 contains two issues that need to be solved to develop a tool that is ready for production work. We saw that productivity with our solution did increase, but the flexibility of the generated applications still requires attention, and especially the tight coupling between database schema and object model. We propose to investigate ways to increase flexibility of these applications, because this is also expected to increase productivity, as we explained in Section 11.1. Another improvement that is expected to increase developer productivity, is to change the traditional way of altering the initially generated source code into a more productive approach.

Two concrete research projects that address these two issues, are framework completion and generation from object-relational mapping files.

**Framework completion** It is worthwhile to investigate a model-driven solution that generates code on frameworks. This approach, where the generation process is suited to fill in the variability points in a framework, is called *framework completion* [25]. The framework contains the functionality that is shared among all applications in the domain, and the remaining application-specific code is generated from a model. We noticed that our tool generates a lot of code that is likely to be shared among several applications—like user interface design and navigation code—and can therefore be captured in a framework. The developer can also make use of the framework when implementing his custom functionality; this is expected to increase his productivity [13]. A future project could, for instance, research code generation on top of a web framework.

**Generate backend from object-relational mapping files** Another interesting project would be to investigate other input models. The implementation and validation phase of this project proved that a database schema offers sufficient information to generate data-driven applications, but the tight coupling between this schema and the object model results in inflexibility. Therefore the participants of our validation session suggested to look into generation of the application backend from mapping files used in an object-relation mapping framework, like NHibernate[1]. These files define how tables in the database map on objects in the application's source code. From this model we can obtain the specified objects and use this information to generate classes for these objects. This results in loose coupling between database tables and objects, and thus better flexibility. With a more flexible generation process, the developer needs less effort to implement his custom functionality.

Secondly, it would be interesting to see how model-driven development can be applied to support development of enterprise-size applications. Would an upgraded version of our solution be able to generate business services and frontends that are part of the service-oriented architecture of an enterprise application? It is of course possible that a different approach is needed for these components in large applications, but in our opinion it is worth to investigate the possibilities.

[1]http://www.nhibernate.org/

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

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

[6] 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.

[7] 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/`.

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

[9] Manfred Broy, Florian Deissenboeck, and Markus Pizka. Demystifying Maintainability. In *WoSQ '06: Proceedings of the 2006 international workshop on Software quality*, pages 21–26, New York, NY, USA, 2006. ACM Press.

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

[11] M. Bruntink and A. van Deursen. An Empirical Study into Class Testability. *Journal of Systems and Software*, 79(9):1219–1232, sep 2006.

[12] Peter Pin-Shan Chen. The Entity-Relationship Model-Toward a Unified View of Data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

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

[17] Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[18] 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`.

[19] 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`.

[20] 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`.

[21] 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.

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

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

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

[25] 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.

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

[27] 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/.

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

[29] Pat Helland. Data on the Outside versus Data on the Inside. In *Proceedings of the 2005 CIDR Conference*, pages 144–153. Published online: http://www-db.cs.wisc.edu/cidr/cidr2005/index.html, 2005.

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

[31] ISO. International Standard ISO/IEC 9126. Information Technology: Software Product Evaluation: Quality Characteristics and Guidelines for Their Use, 1991.

[32] 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.

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

[34] 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.

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

[36] Jesse Liberty and Dan Hurwitz. *Programming ASP.NET*. O'Reilly, 3rd edition, 2006.

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

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

[39] 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`.

[40] 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`.

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

[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] Micheal P. Papazoglou and Willem-Jan van den Heuvel. Service-Oriented Design and Development Methodology. *International Journal of Web Engineering and Technology*, 2(4):412–442, 2006.

[52] 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.

[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] 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.

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

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

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

[66] 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.

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

# Appendix A

# Garage Case Input Models

The database schema used in the Garage Case, is presented in Figure A.1. Listing A.1 and A.2 together form the meta-data file offered to the generation process in the case study.

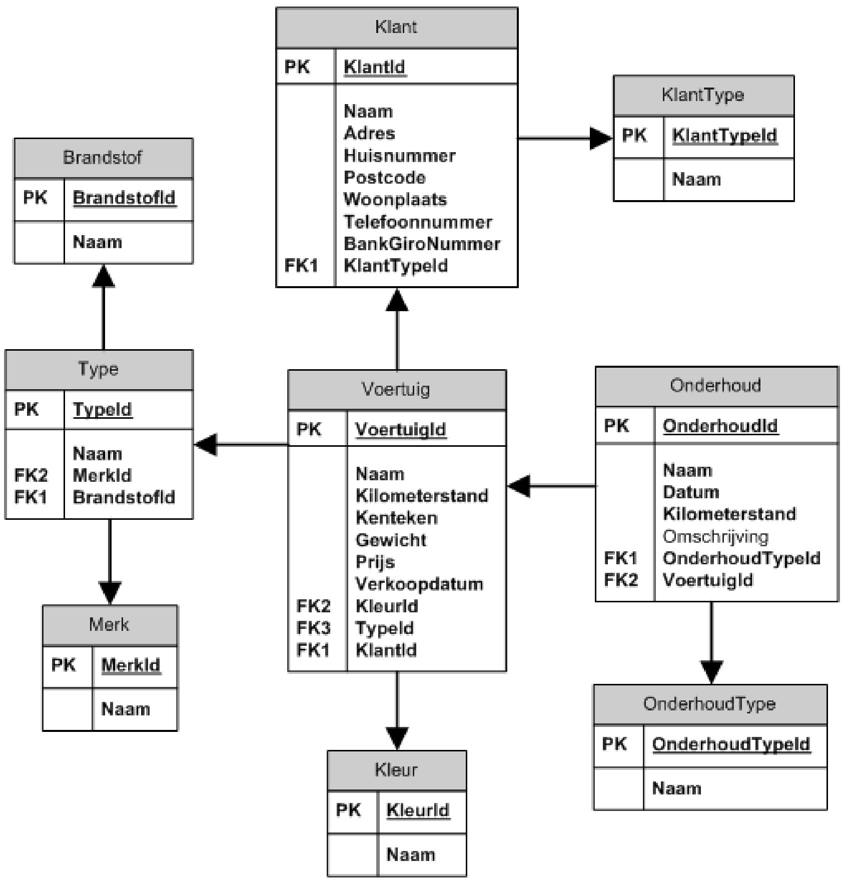


Figure A.1: The Garage Case: Database model

```xml
<?xml version="1.0" encoding="utf-8" ?>
<input>
  <project name="GarageCase"
      rootFolder="C:\GarageCase\Source"
      database="GMSDatabase"
      connectionString="Persist Security Info=False;
          User ID=CRONOS\maartens;Initial Catalog=GMSDatabase;
          Data Source=.\SQLEXPRESS;Integrated Security=SSPI;"/>

  <entities>

    <entity name="Voertuig" plural="Voertuigen" table="Voertuig">
      <views>
        <view name="list">
          <field col="Naam"/>
          <field col="Kilometerstand"/>
          <field col="Kenteken"/>
          <field col="Verkoopdatum"/>
        </view>
      </views>
      <referenceData>
        <ref name="Kleur" plural="Kleuren" table="Kleur"/>
      </referenceData>
    </entity>

    <entity name="Klant" plural="Klanten" table="Klant">
      <views>
        <view name="list">
          <field col="Naam"/>
          <field col="Postcode"/>
          <field col="Woonplaats"/>
          <field col="KlantTypeId"/>
        </view>
      </views>
      <referenceData>
        <ref name="KlantType" plural="KlantTypen" table="KlantType"/>
      </referenceData>
    </entity>
```
Listing A.1: Meta-data file for the Garage Case

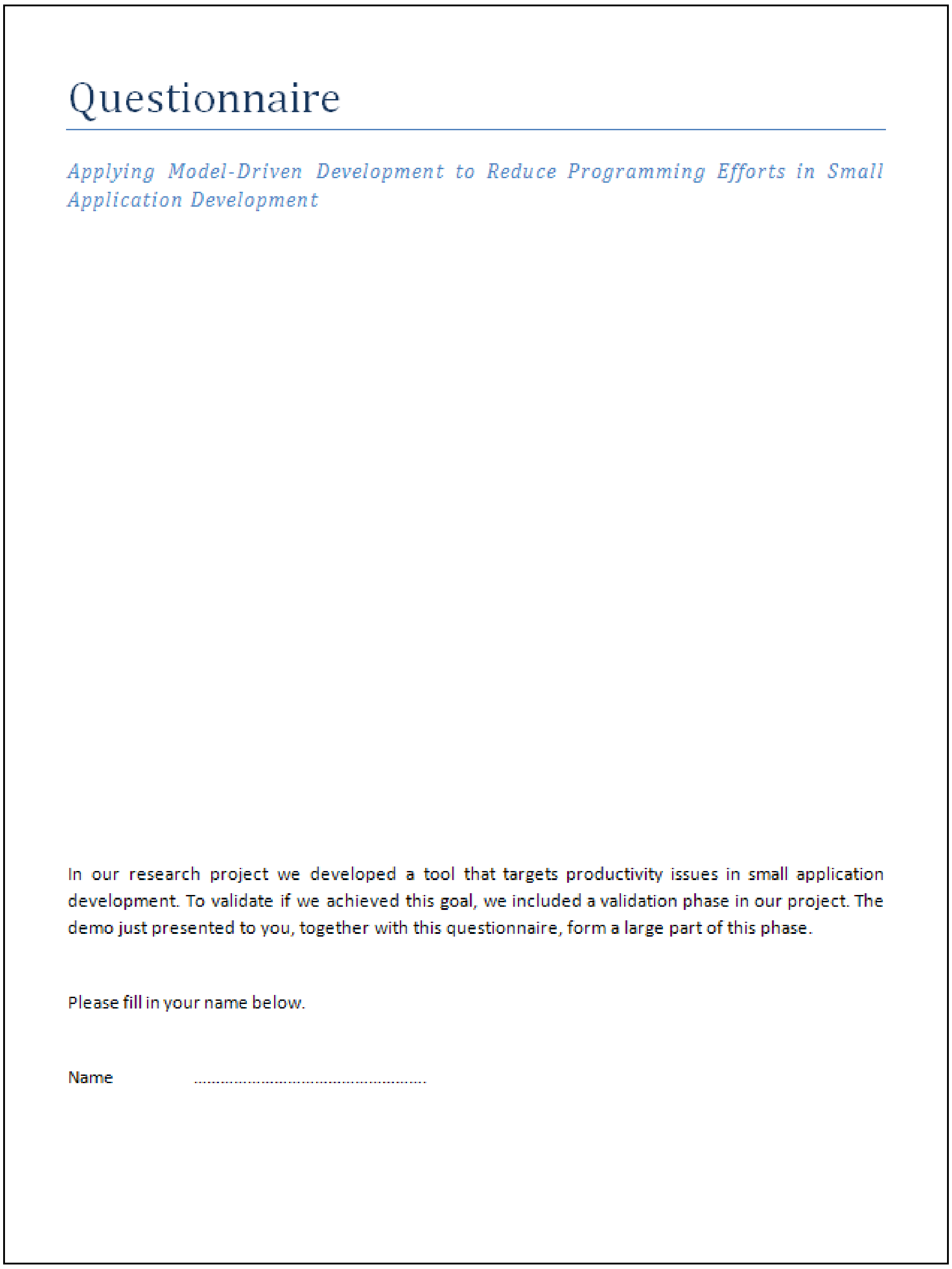
```
    <entity name="Onderhoud" plural="Onderhoud" table="Onderhoud">
      <views>
        <view name="list">
          <field col="Naam"/>
          <field col="Datum"/>
          <field col="OnderhoudTypeId"/>
          <field col="Omschrijving"/>
        </view>
      </views>
      <referenceData>
        <ref name="OnderhoudType" plural="OnderhoudTypen"
            table="OnderhoudType"/>
      </referenceData>
    </entity>

    <entity name="Type" plural="Typen" table="Type">
      <views>
        <view name="list">
          <field col="Naam"/>
          <field col="MerkId"/>
          <field col="BrandstofId"/>
        </view>
      </views>
      <referenceData>
        <ref name="Brandstof" plural="Brandstoffen"
            table="Brandstof"/>
        <ref name="Merk" plural="Merken" table="Merk"/>
      </referenceData>
    </entity>

  </entities>

  <relations>
    <rel one="Voertuig" many="Onderhoud"/>
    <rel one="Klant" many="Voertuig"/>
    <rel one="Type" many="Voertuig"/>
  </relations>

</input>
```

Listing A.2: Meta-data file for the Garage Case (Cont.)

# Appendix B

# Questionnaire

The following pages present the questionnaire we asked the domain experts to fill in after the demo presentation.



Questionnaire

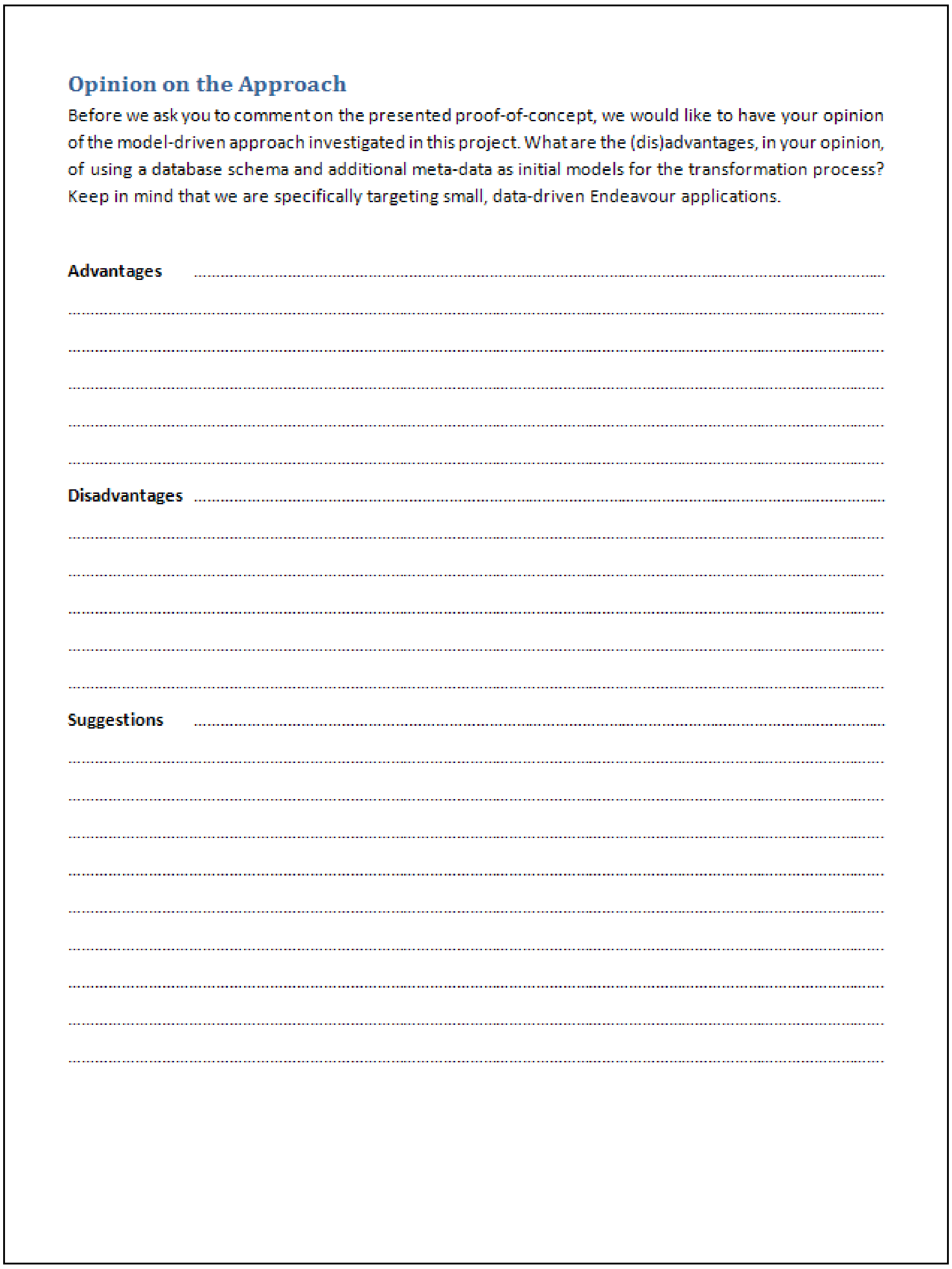
*Applying Model-Driven Development to Reduce Programming Efforts in Small Application Development*

In our research project we developed a tool that targets productivity issues in small application development. To validate if we achieved this goal, we included a validation phase in our project. The demo just presented to you, together with this questionnaire, form a large part of this phase.

Please fill in your name below.

|  |  |
|---|---|
| Name | ........................................................ |

Figure B.1: Page one of the questionnaire.



## Opinion on the Approach

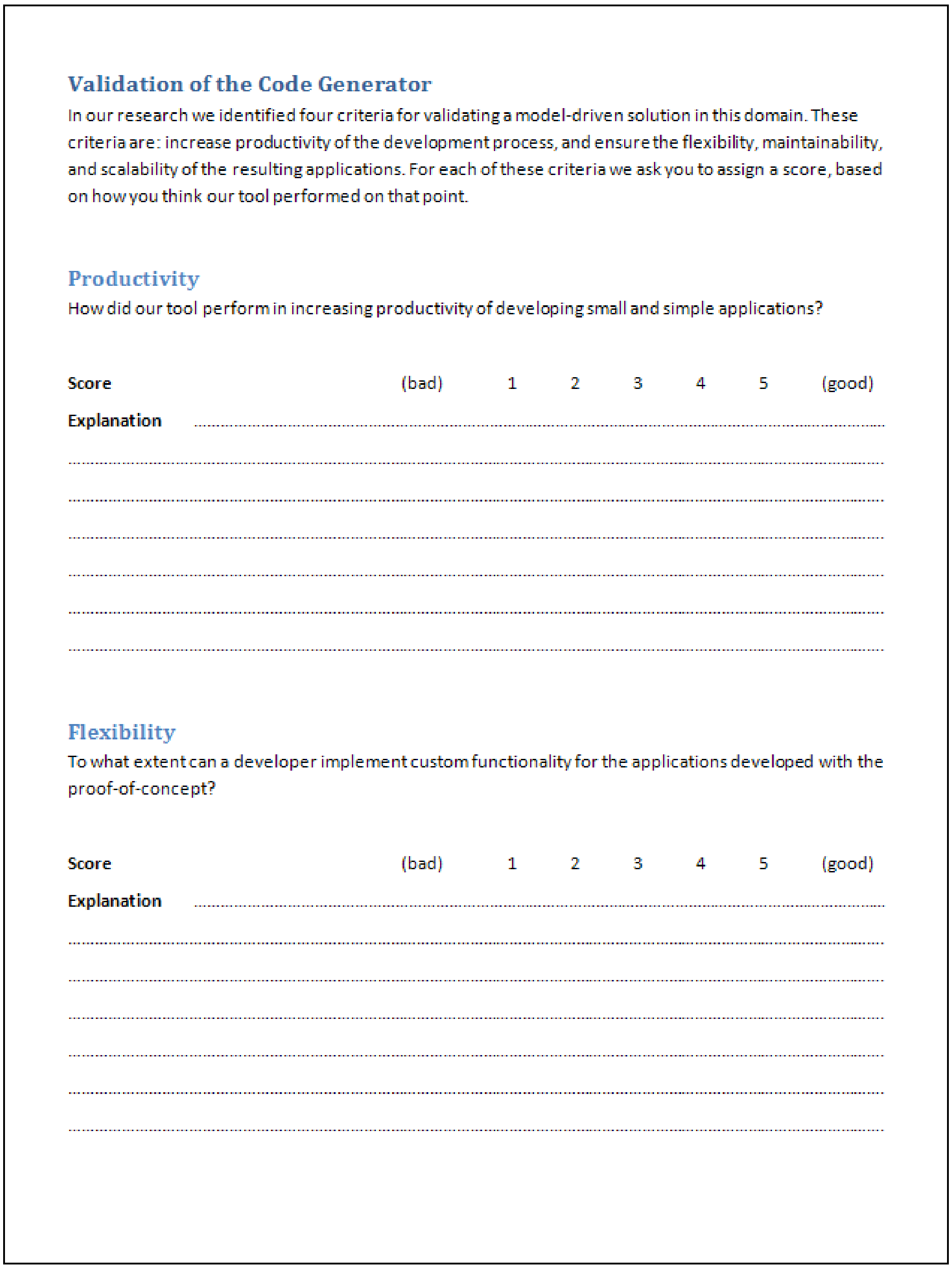
Before we ask you to comment on the presented proof-of-concept, we would like to have your opinion of the model-driven approach investigated in this project. What are the (dis)advantages, in your opinion, of using a database schema and additional meta-data as initial models for the transformation process? Keep in mind that we are specifically targeting small, data-driven Endeavour applications.

**Advantages** ....................................................................................................

**Disadvantages** ....................................................................................................

**Suggestions** ....................................................................................................

Figure B.2: Page two of the questionnaire.



## Validation of the Code Generator

In our research we identified four criteria for validating a model-driven solution in this domain. These criteria are: increase productivity of the development process, and ensure the flexibility, maintainability, and scalability of the resulting applications. For each of these criteria we ask you to assign a score, based on how you think our tool performed on that point.

### Productivity

How did our tool perform in increasing productivity of developing small and simple applications?

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Score** | (bad) | 1 | 2 | 3 | 4 | 5 | (good) |

**Explanation** ........................................................................................................................

........................................................................................................................................

........................................................................................................................................

........................................................................................................................................

........................................................................................................................................

........................................................................................................................................

........................................................................................................................................

### Flexibility

To what extent can a developer implement custom functionality for the applications developed with the proof-of-concept?

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Score** | (bad) | 1 | 2 | 3 | 4 | 5 | (good) |

**Explanation** ........................................................................................................................

........................................................................................................................................

........................................................................................................................................

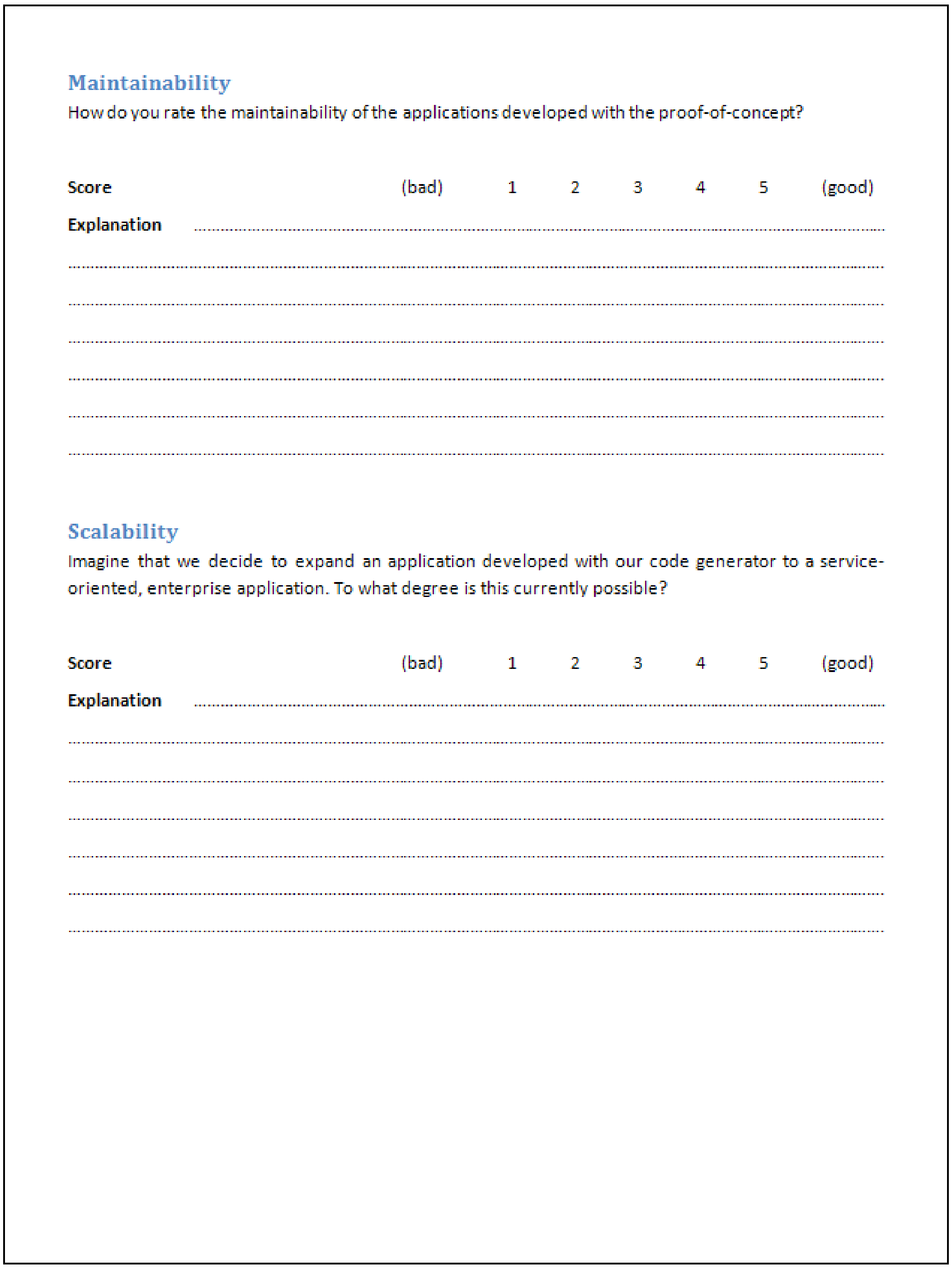
........................................................................................................................................

........................................................................................................................................

........................................................................................................................................

........................................................................................................................................

Figure B.3: Page three of the questionnaire.



**Maintainability**

How do you rate the maintainability of the applications developed with the proof-of-concept?

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Score** | (bad) | 1 | 2 | 3 | 4 | 5 | (good) |

**Explanation** ............................................................

............................................................

............................................................

............................................................

............................................................

............................................................

............................................................

**Scalability**

Imagine that we decide to expand an application developed with our code generator to a service-oriented, enterprise application. To what degree is this currently possible?

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Score** | (bad) | 1 | 2 | 3 | 4 | 5 | (good) |

**Explanation** ............................................................

............................................................

............................................................

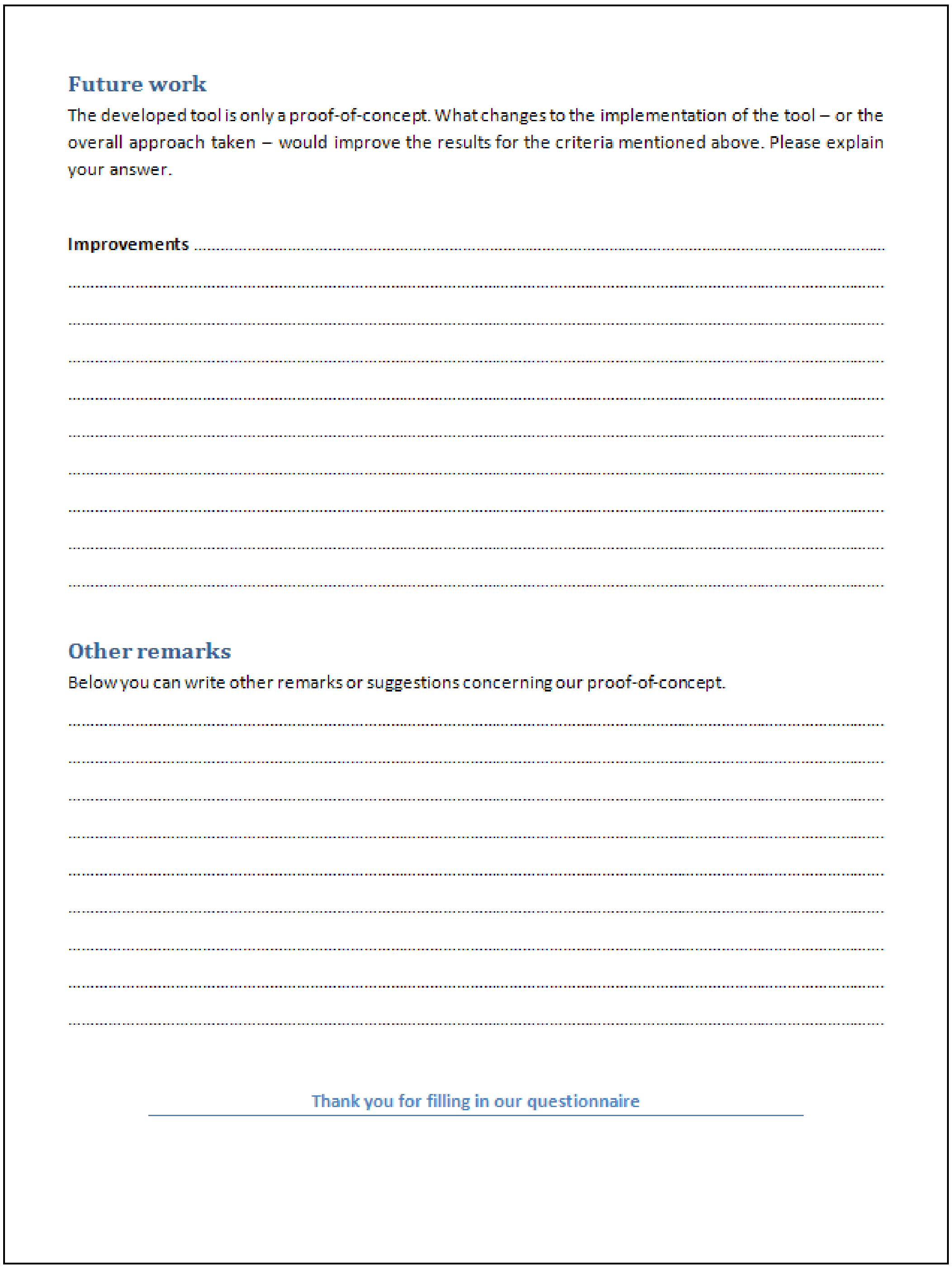
............................................................

............................................................

............................................................

............................................................

Figure B.4: Page four of the questionnaire.



## Future work

The developed tool is only a proof-of-concept. What changes to the implementation of the tool – or the overall approach taken – would improve the results for the criteria mentioned above. Please explain your answer.

**Improvements** ..............................................................................................................................

..............................................................................................................................

..............................................................................................................................

..............................................................................................................................

..............................................................................................................................

..............................................................................................................................

..............................................................................................................................

..............................................................................................................................

..............................................................................................................................

..............................................................................................................................

## Other remarks

Below you can write other remarks or suggestions concerning our proof-of-concept.

..............................................................................................................................

..............................................................................................................................

..............................................................................................................................

..............................................................................................................................

..............................................................................................................................

..............................................................................................................................

..............................................................................................................................

..............................................................................................................................

..............................................................................................................................

Thank you for filling in our questionnaire

Figure B.5: Page five of the questionnaire.