

2019

The Agile Model-Driven Method

Mairon, Klaus

<http://hdl.handle.net/10026.1/15257>

<http://dx.doi.org/10.24382/1239>

University of Plymouth

All content in PEARL is protected by copyright law. Author manuscripts are made available in accordance with publisher policies. Please cite only the published version using the details provided on the item record or document. In the absence of an open licence (e.g. Creative Commons), permissions for further reuse of content should be sought from the publisher or author.



**UNIVERSITY OF
PLYMOUTH**

THE AGILE MODEL-DRIVEN METHOD

by

KLAUS MAIRON

A thesis submitted to the University of Plymouth
in partial fulfilment for the degree of

DOCTOR OF PHILOSOPHY

School of Engineering, Computing and Mathematics

October 2019

Copyright Statement

This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the author's prior consent.

Acknowledgements

In the first place, I have to thank my family for their support during the preparation of this work. Without her patience and consideration, I would not have been able to coordinate the family, profession, and research. Another thanks to my employer, msg systems ag, and my manager Ralf Christmann. Only through the possibilities of reduction to part-time and the understanding of the resulting restrictions has this work become possible at all.

A least as great gratitude my supervisors at the universities in Furtwangen and Plymouth. I am particularly grateful to Prof. Dr. Martin Buchheit for his constant drive and motivation, which was necessary over this long period. Another great thank you to Dr. Shirley Atkinson, Prof. Dr. Martin Knahl, and Prof. Steven Furnell for the patience and support they have shown me. Throughout the years in which I created this work, they always were good counselors and patient listeners to my problems and they gave helpful ideas in joint discussions.

Additionally, I would like to thank my colleagues at msg systems ag as well as my contacts in insurance companies and IT companies who gave me valuable ideas for this work through insights into concrete projects.

October 2019

Author's Declaration

At no time during the registration for the degree of *Doctor of Philosophy* has the author been registered for any other University award without prior agreement of the Doctoral College Quality Sub-Committee.

Work submitted for this research degree at the Plymouth University has not formed part of any other degree either at the University of Plymouth or at another establishment.

Word count of main body of thesis: 36.948 words.

Signed: A handwritten signature in black ink, consisting of several loops and a long horizontal stroke, positioned above a horizontal line.

Date: 10/10/2019

Abstract

The Agile Model-Driven Method

Klaus Mairon, M.Sc.

Centre for Security, Communications and Network Research,

Plymouth University, Plymouth, United Kingdom

Furtwangen Research Node, Faculty of Business Information Systems,

Hochschule Furtwangen University, Germany

Supervisioning: Prof. Dr. Martin Buchheit, Prof. Dr. Martin Knahl,

Dr. Shirley Atkinson and Prof. Steven Furnell

Today the development of business applications is influenced by increased project complexity, shortened development cycles and high expectations in quality [11]. Rising costs in the software development are an additional motivation to improve the productivity by the choice of a suitable development process [59]. In the development of complex applications models are of great importance. Models reduce complexity by abstraction. Additionally, models offer the possibility to build different views onto an application. If models are sufficiently formal they are suitable for the automated transformation into source code. For this reason, an important acceleration and quality factor in the software development is attributed to the Model-Driven Software Development [91]. On the other hand, Model-Driven Software Development requires quite high initial work for the definition of meta-models, domain-specific languages and transformation rules for the code generation process.

A different approach to improve productivity is the use of agile process models like Scrum, Extreme Programming (XP) or Feature Driven Development (FDD) [65]. For these process models an early production of source code and the adjustment of executable partial results are important aspects of the process. The communication with the end user and the direct feedback are the most important success factors for a project and facilitate quick reactions on requirement changes [35]. In agile methods modelling often plays a subordinated role. The requirements will be documented via “user stories” (XP) or “features” (Scrum, FDD). They are summarized either in Product- or Sprint-Backlogs (Scrum) [28][85] or in Feature-Sets (FDD) [24].

From this, the idea is developed to apply agile work practices and techniques in a process tailored to model-driven development. First, existing process models for model-driven development are identified and described. Their common features such as process steps, artefacts and team organisation are worked out and abstracted in a metamodel. The aim is to reuse these process elements in a new agile process model. At the same time, suitable agile practices for modeling are identified, which can support such a process. Additional criteria and suggestions for the improvement of such a process are identified on the basis of case studies from practical model-driven projects.

The Agile Model-Driven Method (AMDM) presents a combination of agile procedures and modelling techniques with the technology of model-driven development. AMDM is iteratively incremental and relies on proven concepts of accepted agile standards [62]. AMDM integrates the development of a domain-specific modelling language, the modelling of problem domains and the development of the software architecture into a context. The development takes place in several cycles of sprints (iterations) which are distinguished in initial sprint, domain sprint and value sprint. Parallel to the development

of domain language and application, the software architecture is developed evolutionarily and transferred to development.

Finally, based on the mentioned case studies from the practice and investigations of model-driven projects in other industrial companies and business fields is checked how AMDM can contribute by agile concepts to increase efficiency in model-driven projects and how the expressed criticisms and problems from these studies can be avoided.

Table of Contents

1	Introduction.....	14
1.1	Research Background	14
1.2	Objectives of this Work	16
1.3	Applied Methodologies	17
1.4	Summary	19
2	Model-Driven Development	23
2.1	Software Engineering and Models.....	23
2.2	Definitions	24
2.2.1	Model, Platform and View	25
2.2.2	Model-Based Development vs. Model-Driven Development	27
2.2.3	The Term “Domain Architecture”	28
2.3	Methodologies for Model-Driven Development	29
2.3.1	ODAC	30
2.3.2	MASTER	32
2.3.3	DREAM.....	35
2.4	Critical View / Problems in MDD Projects	38
2.4.1	Effort/Cost Drivers in MDD Projects	38
2.4.2	Limitations of Model-Driven Development	39
2.4.3	General Limitations and Criticisms	42
2.5	Summary	43
3	Agile Development and Modelling – Existing Approaches	44
3.1	The Agile Approach.....	45
3.1.1	Agile Principles	47
3.1.2	Agility	48
3.1.3	Agile Techniques and Practices	48
3.2	Opportunities and Limitations of Agile Approaches.....	50
3.3	Agile Methods for MDD Support.....	53
3.4	Existing Approaches, Solved Problems and Limitations.....	58
3.4.1	Agile Model Driven Development (AMDD).....	58
3.4.2	MIDAS Framework	61
3.4.3	Feature Driven Development (FDD)	63
3.5	Consequences	66
3.6	Summary	68
4	Case Studies – Practical Experience in MDD Projects	69
4.1	Case Study 1: Interfaces to Legacy Systems	70
4.1.1	Initial Situation	70
4.1.2	MDD Approach	70
4.1.3	Result / Experience	74
4.2	Case Study 2: Software Component Development.....	75
4.2.1	Initial Situation	75
4.2.2	MDD Approach	76
4.2.3	Result / Experience	78

4.3	Case Study 3: Insurance Programming Language	79
4.3.1	Initial Situation	79
4.3.2	MDD Approach	79
4.3.3	Result / Experience	80
4.4	Experiences from other Case Studies	80
4.4.1	ABB Robotics and Ericsson	81
4.4.2	Autoliv, Sectra und Saab Aerospace.....	82
4.4.3	IBM.....	84
4.4.4	Motorola	85
4.5	Summary.....	86
5	Approach: Using Agile Elements in MDD-Processes	88
5.1	Combination of MDD and Agile	88
5.2	A detailed view on MDD methodologies	89
5.3	Commonalities of the Reviewed Methods.....	90
5.3.1	Project Phases and Steps.....	92
5.3.2	Artefacts and Result Types	94
5.3.3	Roles and Team Members	97
5.4	Agile Techniques and Practices for Modelling.....	99
5.4.1	Assume Simplicity / Simple Design.....	100
5.4.2	Architecture Envisioning	101
5.4.3	Model Storming	102
5.4.4	Just Barely Good Enough	102
5.4.5	Iteration Modelling	103
5.4.6	Multiple Models.....	104
5.4.7	Document Continuously	105
5.4.8	Some other Practices.....	106
5.5	The Appropriate Agile Development Process	108
5.6	The Meaning of Architecture in Agile Projects.....	110
5.6.1	Architecture - a Definition.....	111
5.6.2	The Difference between Architecture and Design.....	114
5.6.3	Agile Best Practices for Architecture	115
5.7	Known Limitations	119
5.8	Summary.....	120
6	The Agile Model-Driven Method: AMDM	123
6.1	Requirements and Constraints	124
6.2	Findings from the Studies	125
6.3	Definitions	127
6.3.1	Team and Role	127
6.3.2	Backlog	127
6.3.3	Iteration.....	128
6.3.4	Architecture	128
6.3.5	Domain-specific Language (DSL) and Metamodel.....	129
6.3.6	Transformation and Transformation Rules.....	130
6.4	Basic Concepts.....	130
6.4.1	Teamwork	130
6.4.2	Evolutionary Software Architecture	132
6.4.3	Backlog Content	135

6.4.4	Modelling Language (DSL).....	135
6.4.5	Modelling in the Development Process.....	137
6.5	Implementation.....	138
6.5.1	Team Members and Roles.....	138
6.5.2	Artefacts.....	141
6.5.3	Process Steps.....	143
6.5.4	Communication.....	148
6.6	Summary.....	150
7	Evaluation.....	151
7.1	Significance for the Case Studies.....	151
7.1.1	Case Study 1: Interfaces to Legacy Systems.....	151
7.1.2	Case Study 2: Software Component Development.....	152
7.1.3	Case Study 3: Insurance Programming Language.....	153
7.1.4	Other Case Studies.....	153
7.2	Pilot Project.....	157
7.2.1	Goals of the Project.....	157
7.2.2	Team.....	159
7.2.3	Project Course.....	159
7.2.4	Project Experience.....	163
7.3	Agile Review.....	164
7.4	Summary.....	166
8	Conclusions.....	168
8.1	Achievements.....	168
8.2	Limitations.....	169
8.3	Suggestions for Future Research.....	170
	References.....	171
	Appendix: Interviews from the Pilot Project.....	183
A1:	Interview Product Owner.....	183
A2:	Interview Domain Architect/Domain Developer.....	184
A3:	Interview Business Analyst.....	186
A4:	Interview Application Architect.....	188
A5:	Application Developer 1.....	190
A6:	Application Developer 2.....	191
	Publications.....	192
	Biographical Information.....	193

List of Figures

Fig. 1: Dependencies between the parts of the Domain Architecture [91].....	29
Fig. 2: The ODAC Process [7]	31
Fig. 3: The MASTER Process [7].....	33
Fig. 4: The DREAM Process [7]	36
Fig. 5: The two Phases of an MDD Project	39
Fig. 6: Hype Cycle for Application Development, Gartner, 2008.....	44
Fig. 7: Agile Techniques and Practices	50
Fig. 8: Characteristics of the selected Methods (Excerpt from [78]).....	54
Fig. 9: Characteristics of the AMDD Methodology (Excerpt from [78]).....	56
Fig. 10: Awareness Level and Distribution of Agile Methodologies (see [78]).....	57
Fig. 11: The AMDD Process [5].....	58
Fig. 12: Model Driven Architecture of MIDAS [20].....	61
Fig. 13: Modelling within the FDD Process [6]	64
Fig. 14: Dependencies between Artefacts in MDD Projects	67
Fig. 15: Models and Generated Artefacts in Case Study 1	71
Fig. 16: Sample UML Class Diagram for the Definition of a COBOL Copybook	72
Fig. 17: Sequence in the Activity Diagram refined in the State Diagram	73
Fig. 18: Project Effort in Case Study 1	74
Fig. 19: Parts of the Software Framework in Case Study 2.....	75
Fig. 20: Metamodel for the Description of Software Components.....	77
Fig. 21: Project Effort in Case Study 2	78
Fig. 22: Process Steps in MODA-TEL and the corresponding Element in the Metamodel	90
Fig. 23: Metamodel with Process Elements	91
Fig. 24: Metamodel: Roles in Existing MDD Processes	92
Fig. 25: Artefacts in the Model-Driven Development.....	95
Fig. 26: Identified Project Member Roles in MDD-Projects.....	97
Fig. 27: Agile Modelling Principles	100
Fig. 28: Document Continuously	105
Fig. 29: Important Agile Methods [62].....	109
Fig. 30: UML, Meta-meta-Models and Profiles	129
Fig. 31: Team Structure in AMDM	131
Fig. 32: Joint Development of Architecture (“Big Picture”).....	134
Fig. 33: Class Archetypes and Typical Associations [24].....	136
Fig. 34: AMDM Process Overview	144
Fig. 35: Domain Sprint	146
Fig. 36: Value Sprint.....	147
Fig. 37: Microservices in the Context of the existing Back-Office Solution	158

Glossary

AMDD	Agile Model Driven Development (agile modelling technique)
AMDM	The Agile Model-Driven Method
CIM	Computation Independent Model
CWM	Common Warehouse Metamodel (OMG standard)
DSL	Domain Specific Language
DSDM	Dynamic System Development Method (agile methodology)
EDOC	Enterprise Distributed Object Computing (UML-profile, OMG)
FDD	Feature-Driven Development (agile methodology)
ISO	International Organization for Standardization
IST	Information Society Technology
ITU-T	International Telecommunication Union (Telecommunication Standardization Sector)
MDA	Model-Driven Architecture (OMG standard)
MDD	Model-Driven Development
MDE	Model-Driven Engineering
MDSD	Model-Driven Software Development
MOF	Meta Object Facility (OMG standard)
OMG	Object Management Group
PIM	Platform Independent Model
PLE	Product Line Engineering
PSM	Platform Specific Model
RM-ODP	Reference Model of Open Distributed Processing

QVT	Query/Views/Transformations (OMG standard)
TDD	Test-Driven Development (agile methodology)
UML	Unified Modelling Language (OMG standard)
XMI	XML Metadata Interchange (OMG standard)
XP	eXtreme Programming (agile methodology)

1 Introduction

1.1 Research Background

In the IEEE Standard Glossary of Software Engineering Terminology [55] “software engineering” is defined as “(1) *The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).*” But the discipline of software engineering holds two key challenges that separate it from other engineering disciplines. In their paper about the impact of agile methods on software project management, the authors Coram and Bohner explain these challenges as follows: “*Software, a conceptual and often intangible product, changes and evolves at a much higher rate than integrated circuits or steel. While software is a changeable product, there is an increased cost the later in a project the change occurs.*” [29][51] The authors use this statement to describe the challenge of accepting changes in the requirements for a software system, while at the same time limiting the costs of the project, which are burdened by late changes. Managing change while reducing the impact on project costs is a key challenge for project management. And it is also a challenge for the software development process.

In the last three decades, a large number of software development processes have been introduced in software development. In the hope of making the development process more efficient, there were always new technologies, programming paradigms, languages and methods. On the one hand, the different methods and development processes try to counteract the increasing complexity of the development of information systems by consistently defining and structuring the necessary tasks. So Scacchi outline in [83] that

“software process models often represent a networked sequence of activities, objects, transformations, and events that embody strategies for accomplishing software evolution.” But a study from 1999 [70] claims, that the analyzed software development methodologies are “too mechanistic to be used in detail”. The study concluded, that at this time industrial software developers become skeptical about “new” methodologies. According to Abrahamsson et al., this is the background for the emergence of agile software development methods in the late 1990s [1]. Also Dingsøy et al. state in [33] the “agile development methods as a reaction to plan-based or traditional methods.”

On the other hand, there was an attempt to industrialize software development and bring more automation into the development process. The use of models in the first computer-aided software engineering tools (CASE tools) and the emergence of structured methods were an approach to consider graphical models as part of the software design and to create standardized documentation [89]. These tools and standardized models also made it possible to automate the software development process, such as generating code from graphical models or user interfaces from a graphical user interface description. According to the structured methods of the 1970s and 1980s, the standardization of the different modeling languages by the UML (Unified Modelling Language) was the basis for model-driven development and, in addition, model-driven engineering [84][89]. With the standardization of model-driven development by the Object Management Group (OMG), the automated generation of software has gained widespread acceptance.

The OMG Standard MDA (Model-Driven Architecture) is the foundation for uniform software development based on (partly) automated model transformations. The associated standards MOF (Meta Object Facility), CWM (Common Warehouse MetaModel) and XMI (XML Metadata Interchange) are still used today in various tools and code

generators. Based on the UML, a large number of UML profiles have been developed to describe different functional and technical requirements. These domain-specific modeling languages are also called UML dialects. However, the multitude of these dialects and the associated softening of the UML standard is one of the criticisms of this approach.

In today's modern industrial software development, the pure MDA standard is only occasionally used. Nevertheless, individual components of the MDA are applied. Modeling tools and code generators exchange model information via the XMI standard, and the modeling language is often based on UML and MOF. It can be observed that the basic technical structures of the defined application architecture are generated on the basis of model information, but not so much the functional logic of the application systems. In an initial step, the starting point for further development is created. Model-driven development is therefore not embedded in the respective process model. And this is where this thesis comes in.

1.2 Objectives of this Work

Model-driven development enables the technical aspects of the software architecture to be regarded, further developed and adapted separately from the business functionality. The aim of this work is to support the model-driven development of modern business applications through an optimized agile process model. It is particularly important to consider that the principles of agile software development, such as the early delivery of partial results, must also be recognizable in the new process model.

In addition, the new process model should not only define a process, but also the roles involved in the process and how they interact with each other. The individual tasks of these roles must be defined. A separation between technical tasks and functional content

is particularly important for model-driven development [91]. This is a major advantage of model-driven development, as technical aspects can be developed and adapted separately without influencing business models. This must also be reflected in the roles' distribution of tasks.

Because model-driven development brings additional complexity to development, another goal must be to reduce the complexity or make it easier to handle.

For the definition of the new process model, it must be checked in advance to what extent process models for model-driven development exist at all. These must be examined to determine which phases and development cycles they include for model-driven development in particular and which they may have in common. In addition, it must be checked to what extent they already take agile principles into account or which sub-elements of these process models are suitable for use in an agile environment.

Finally, the new process model should address small and medium-sized teams, which is the common practice in business application development today [30][45]. Furthermore, elements of existing agile process models should be reused to facilitate the introduction and acceptance of a new process model.

1.3 Applied Methodologies

Evaluating the design of a software development process is difficult. This also applies to this work. In this environment, quantitative data can only be used to make statements about the distribution and use of software development processes. In this work, sources

such as Gartner, Forrester Research and the German Research Center for Computer Science (FZI) are used.

Therefore, qualitative research methods are used to assess the development process defined in the thesis. In the development of the fundamentals, a qualitative content analysis based on case studies from practice is applied. These case studies come partly from the author's concrete working environment and partly from published case studies from projects of IBM, Motorola, ABB Robotics and others.

In the analysis of existing development processes for model-driven development, a methodology is applied that Dr. Susanne Strahinger describes in her dissertation at Darmstadt Technical University [95]. In this method, the elements of the processes are described and categorized using metamodels so that differences and similarities can be better identified.

For the evaluation of the defined development process, a projection on the case studies mentioned above takes place first. Here the influence of the new development process on the problem areas described there is rated. In addition, a pilot project in the author's working environment will test the concrete use of the development process. The assessment of the development process on the basis of this pilot project is carried out using the qualitative empirical research method of interviews with selected participants of the project. In a third section, the extent to which the newly defined development process implements the principles of the agile manifesto is examined.

1.4 Summary

This research is to be classified in the field of software engineering and examines how model-driven development can be optimized with agile principles and techniques and thus used more efficiently. To this end, existing methods and development processes for model-driven development are first of all identified and examined for their common features, strengths and weaknesses. It is also analyzed to what extent models play a role in agile software development. The question arises as to which solutions already exist for the support of model-driven development in agile process models and which consequences result from this. Using reports and case studies from industry, problems and opportunities of model-driven development are identified. These case studies describe experiences from industrial projects that have led to positive and negative assessments of model-driven development. They form the basis for the subsequent evaluation of new agile process elements and the assessment of the extent to which these process elements could improve the respective project situation.

The solution approach of this thesis consists of two elements: Firstly, the basic structure of process models for model-driven development. These are identified in the investigated process models and abstracted and summarized by means of a metamodel. On the other hand, from a study of existing agile techniques for modeling in general. Based on these two elements, the Agile-Model-Driven Method is defined as a new process model for model-driven development. Finally, this process model is projected onto the case studies and evaluated with regard to its possible impacts on these projects.

This thesis is structured as follows: Chapter 2 - Model-Driven Development - begins with an overview of model-driven software development. Starting from the concept of the model in modern software engineering, the chapter leads to the MDA as a standard

approach for model-driven software development. In addition to defining the relevant terms and explaining their meaning for work, the chapter contains a summary of existing process models and methods in the area of model-driven software development. It is evident that none of the existing - above all academic - approaches have been accepted in practice. Nevertheless, the common elements of these approaches provide an indication of which elements should be considered in a future agile model-driven process. Examples of this are the definition of the technical framework for the development, the collection of requirements or the handling of platform-independent or platform-specific models.

Chapter 3 - Agile Development and Modelling - Existing Approaches - first explains the basic principles of agile software development. In addition to a consideration of the possibilities and limitations of agile software development, the focus of this chapter is on the identification of agile practices and process models that seem appropriate to support model-driven projects. In the following, it deals with the closer examination of the identified agile practices and existing academic approaches for agile MDD. Because MDD is characterized also as an architecture-centric approach, the chapter includes a discussion of the agile approach to architecture and design. Finally, there is a consideration of consequences of the agile approach to MDD and the impact on the individual partial results in a model-driven project.

Chapter 4 - Case Studies - Practical Experience in MDD Projects - examines the use of model-driven software development in industrial software development practice. On the basis of reports and case studies from industry, positive and negative experiences with model-driven software development are collected. The three main case studies come from the author's project environment. These are complemented by further case studies of renowned companies such as ABB Robotics, IBM, Motorola and others.

Chapter 5 - Approach: Using Agile Elements in MDD-Processes - describes the basic idea of adding agile elements to processes for model-driven development. The aim is to ensure that the typical high initial effort of model-driven development is better spread over the project phases and that project results can be made available to the end customer earlier. This raises the question of which process elements are needed for model-driven development. This applies not only to process phases and steps, but also to other elements such as important partial results (artefacts) and roles of the project participants. Based on the examined process models for model-driven development, common features are identified and abstracted in a metamodel. This forms the basis for a newly defined agile process model.

If agile process models and model-driven development are to be successfully combined, it is necessary to consider agile working techniques for modeling. For this purpose, first of all, different agile modeling techniques are examined to determine whether they are suitable for use in model-driven development. Subsequently, agile process models are considered which contain models as relevant partial results and can thus serve as a starting point for an adapted process. To this end, the extent to which modeling and architecture are anchored in these process models will be investigated. It is also interesting to see which approaches are most widely accepted in practice, as this is a good starting point for a new process. Finally, this chapter examines the aspect of architecture in the context of agile projects. The background is the focus on architecture as the basis for the model-to-code transformation required by model-driven development. The chapter concludes with a reflection on known limitations of agile modelling.

Chapter 6 - The Agile Model-Driven Method: AMDM - deals with the definition of the Agile Model-Driven Method. This method combines agile working techniques for use in model-driven development. A new approach is defined on the basis of existing process models or their elements. The starting point is the metamodel defined in chapter 5. The chapter begins with a summary of the findings from chapters 2-5, followed by the definition of important terms such as team, role, architecture, etc. in the context of AMDM. Subsequently, the underlying concepts such as evolutionary architectural development are explained. The chapter concludes with a description of the process and the roles of the project members, artefacts and process steps in the interaction.

It is difficult to evaluate a new development process. Chapter 7 - Evaluation - attempts to do this using the case studies presented in Chapter 4. The approach is to assess the impact of AMDM on the above-mentioned criticisms and problems, thus enabling an assessment of the process. In addition to this consideration, AMDM is tested as a new development process in a concrete project scenario in the author's working environment. The experiences gained from this project are included in the evaluation. Finally, the principles of the agile manifesto are used to determine whether AMDM meets the criteria of an agile approach.

Chapter 8 summarizes the results of this work and shows the limitations of the approach or methodology AMDM. In addition, references are given to further research work or related research areas.

2 Model-Driven Development

2.1 Software Engineering and Models

The use of models for the development of complex business applications is not a new approach. As is well known, the mapping of software system requirements by models was generally accepted even before the development of UML. CASE should facilitate the creation of software and support as many phases of software development as possible. In the following, models for defining requirements or mapping system behavior in software development became more and more relevant, and the popular use of the standardized UML finally enabled the approach of model-driven development [89].

In the first decade of the 21st century, model-driven development (MDD) and model-driven engineering (MDE) became increasingly important for professional software development and science. Model-driven development supports the work with models on different abstraction levels [89]. It supports automated transformation between these levels with the ability to generate source code or other development artefacts. With this approach, model-driven development pursues the objectives of interoperability, portability, productivity and quality improvement as well as the reuse of business models [40][84][91].

The Model-Driven Architecture (MDA) is the standardized approach of the Object Management Group (OMG) for MDD and is based on the following OMG standards: The Unified Modelling Language (UML), XML Metadata Interchange (XMI), Meta Object Facility (MOF) and Common Warehouse Metamodel (CWM) [40][68]. According to [68] MDA is a model-driven approach, *“because it provides a means for using models to direct the course of understanding, design, construction, deployment, operation,*

maintenance and modification". Another important concept of model-driven development is the concept of domain-specific languages. A domain-specific language (DSL) can be expressed by graphical symbols or texts and is used to describe the concepts of a particular domain. A DSL must be defined via a metamodel (e. g. based on MOF), including its static semantics and a corresponding concrete syntax [91]. The MDA standard approach to defining a domain-specific language is based on extending UML by defining UML profiles [40], but model-driven development does not depend on it. In [96] the authors name Excel tables, ASCII text and source code as alternative representations of a model. The authors explain that there are many different types of models used for model-driven development in industrial software development.

Another central theme of model-driven development is the transformation of models. Transformations describe how model elements are to be mapped to the next abstraction level, be it another model or source code. For this purpose, the OMG provides the Meta Object Facility (MOF) with its four abstraction levels for metamodels (called M0 to M3)[40]. For standardized model transformations, OMG has defined Query/Views/Transformations (QVT), a programming language for describing transformations. This is a very complex specification, which is only supported by a few tools, so that in practice many other transformation frameworks are used [91][96].

2.2 Definitions

The following terms are important in the context of Model-Driven Development (MDD) and Model-Driven Architecture (MDA) as well as in the context of this work.

2.2.1 Model, Platform and View

- **Model:** Miller et al. define a model in [68]: “*A model of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modelling language or in a natural language.*” Or simply like in [40]: “*An abstraction of a system.*” And Stahl et al state in [91] “*a model is an abstract representation of a system’s structure, function or behaviour.*” For model-driven development, models are initially relevant for the description of domain-specific languages (DSL) and their structure, and secondly for the description of the functional and non-functional requirements of an application system using models, again using one or more defined DSLs. The language elements of a DSL can be extensions of an existing modeling standard (e. g. UML), or they can consist of graphical symbols or texts derived from the corresponding problem domain. Both types of models are relevant for this work.
- **Metamodel:** Frankel define a metamodel as “*a model of the constructs that make up a language*” [40]. And Miller et al. explains in [68] that, “*in language specifications the abstract syntax of the language is specified as a MOF-compliant metamodel*”. A domain-specific language is often described using an appropriate metamodel. Also in the context of this work, it is assumed that a metamodel is the basis of a domain-specific language and that the development of the metamodel represents a significant step in the context of an MDD project.
- **Model transformation:** According to Miller et al. “*model transformation is the process of converting one model to another model of the same system.*” [68]. For model-driven development, this means either the transformation of an abstract

model into a more specific model (M2M: Model to Model Transformation) or the transformation of a model into text or source code (M2T: Model to Text Transformation). A transformation can be performed manually or automatically. The enrichment of a so-called platform-independent model (PIM) with elements of a DSL (e.g. with annotations in the form of stereotypes and tagged values) is a common way of a manual M2M transformation into a so-called platform-specific model (PSM). The further transformation of a PSM into the source code, an M2T transformation, is often automated by a code generator in a model-driven project.

- **Platform:** The term “platform” is described in [68] as *“a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specific usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented.”* Frankel summarizes this in [40] as *“a specified technology or set of technologies”*. In the following, the term platform always refers to a defined target environment for which specific models, documents, source code or other artefacts are generated based on a specific domain-specific language.
- **Viewpoint / View:** Miller et al. defines in [68] a viewpoint on a system based on [56] as *“a technique for abstraction using a selected set of architectural concepts and structuring rules, in order to focus on particular concerns within that system.”* And they further explain *“the Model-Driven Architecture specifies three viewpoints on a system, a computation independent viewpoint, a platform independent viewpoint and a platform specific viewpoint.”* A view of a system or a viewpoint model is defined as *“a representation of that system from the perspective of a chosen viewpoint”* [68]. The models required for a model-driven

development project can be divided into two main views. On the one hand, there is the architect's perspective with the metamodel for the domain-specific language, its elements and the associated transformation rules. On the other hand, there is the view of the development team, which uses the defined domain-specific language to model the requirements of an application. Creating both views is relevant for the execution of a model-driven project.

2.2.2 Model-Based Development vs. Model-Driven Development

In model-driven development, models "drive" the process and are the most important artefacts of development. Models also play an important role in model-based development, but are not essential for the process. An example of model-based development would be a software development process in which Business Analysts specify system models, but developers manually write the code based on these models (no automatic transformation, no code generation). This research deals with model-driven development and thus the automatic generation of artefacts including the source code.

According to [60][69][84], Model-Driven Engineering is an approach to software development in which models, not programs, are the main output of the development process. MDE and MDA are often considered equivalent. In [89] Sommerville, however, says that MDE has a broader scope than MDA, because "MDA focuses on the design and implementation phases of software development, while MDE deals with all aspects of the software engineering process". The focus of this work is on model-driven software development in the understanding of the MDA.

2.2.3 The Term “Domain Architecture”

Asadi et al. mention several relevant artefacts for model-driven development in their paper [8]. These are different types of models or specifications (e. g. PIM, PSM), generic frameworks, code and transformation specifications. In [91], the authors also mention a reference model and a reference implementation as relevant artefacts. They also define the term "Domain Architecture", which summarizes the infrastructure of an MDD project and the associated artefacts. It consists of the following three components:

- **Domain:** This term includes the elements needed to describe a problem domain using models. First of all, this is the metamodel used to define the domain-specific language. There is also a DSL editor and a reference model. The reference model demonstrates the use of the domain-specific language using an example.
- **Transformations:** This component contains the definition of the necessary model transformations. These are derived from a reference implementation based on the corresponding reference model (see Domain). The reference implementation is also the basis for defining the required programming model.
- **Platform:** The third part of the Domain Architecture contains all the components on which the generated code is based. This includes code generators, runtime environments of the target programming language, libraries, etc. According to [91], the platform supports the implementation of the domain with the aim of making the transformation of formal models as simple as possible.

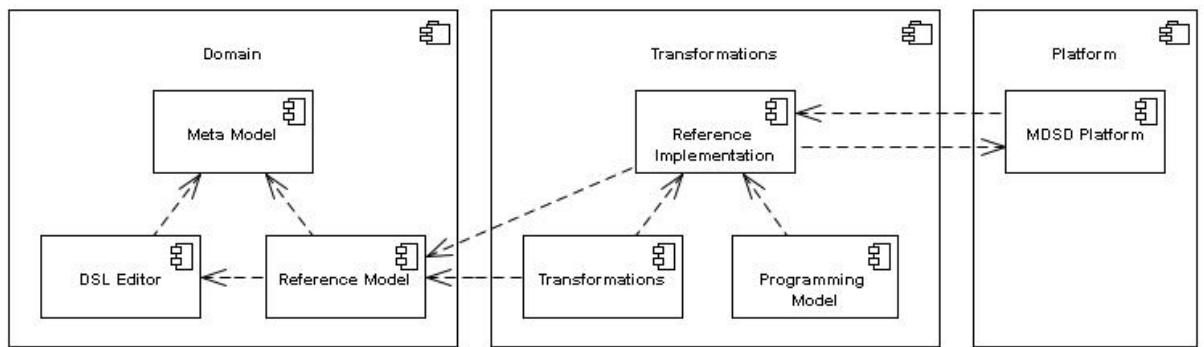


Fig. 1: Dependencies between the parts of the Domain Architecture [91]

Fig. 1 shows the dependencies and influences between the three components of the Domain Architecture. It makes it clear that the transformations depend on the elements of the domain on the one hand. On the other hand, however, they are also influenced by the platform and the technologies used. The development of the Domain Architecture in an initial project phase is therefore an important and necessary, but also time-consuming and expensive step in an MDD project. Therefore, it will be interesting to see which process models are available for model-driven development and whether this initial development step is reflected in these models.

2.3 Methodologies for Model-Driven Development

Although model-driven development has had a major impact on the development of software and various surveys such as [94] or [97] conclude that the promised benefits are achieved, the OMG-standard MDA does not define a corresponding software development process or methodology. According to [8] a software development methodology (SDM) is “a framework for applying software engineering practices with the specific aim of providing the necessary means for developing software-intensive systems”. The authors explain “a methodology consists of two main parts: a set of modelling conventions comprising a modelling language (syntax and semantics), and a

process which provides guidelines about the order of the activities and specifies the artefacts developed using the modelling language”.

In practice, however, there is no established methodology for MDD projects. But in academic research, several methods are defined. These methods are: the ODAC methodology [43][44], MASTER [64], C³ [52], DREAM [90], MODA-TEL [42] and DRIP-Catalyst [46].

Three of these methods and their main features are described below. Together, they provide a good overview of the necessary activities and practices in the development process of an MDD project. The aim is to identify similarities and differences in methodology. This can be used to derive important process steps, artefacts and roles that an MDD process must implement.

2.3.1 ODAC

ODAC is a project with the aim to simplify the modelling of open and complex distributed applications. The project is based on the ODAC Reference Model of Open Distributed Processing (RM-ODP) [57][79]. RM-ODP was developed by the ISO and the ITU-T, and includes a reference model, as well as an architectural framework for the development of distributed applications. The core concept of RM-ODP is the notion of viewpoints, on which the modelled views are based. The viewpoints enable to structure the modelling activities. RM-ODP is primarily an architectural framework and not a method.

Based on RM-ODP, ODAC defines processes and process steps for the development of distributed applications and uses the UML notation as well as the mechanism of UML

profiles. In addition, project guidelines for software developers and system architects were developed and described within the ODAC framework. ODAC is not restricted to a specific application domain, but since today's business applications are often developed as distributed multi-tier applications, the RM-ODP-based ODAC methodology seems to be a possible development methodology, especially in this environment. For this reason, ODAC is also very interesting in the broader context of this work.

ODAC is based on the RM-ODP-based concept of viewpoints and defines the necessary activities, from analysis to design and implementation. The Enterprise / Information and Computational views are assigned to the analysis phase, the Engineering view of the design phase, and the Technology view to the implementation.

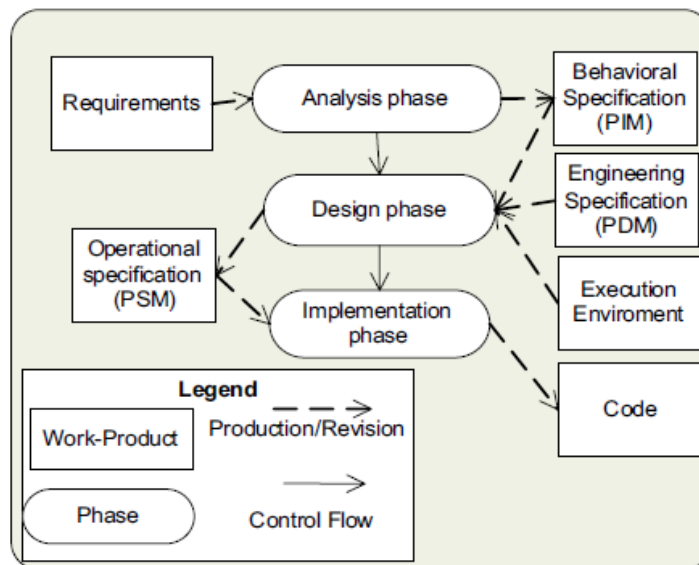


Fig. 2: The ODAC Process [7]

Results of the phases are the Behavioral Specification (analysis), the Engineering Specification (design) and the Operational/Technical Specification (implementation):

- **Behavioral Specification:** Modelling of functional behavior and functional requirements.
- **Engineering Specification:** Description of the platform with the necessary tools, libraries, etc.
- **Operational/Technical Specification:** Mapping instructions for the transformation of the behavioral specification to the target platform (configuration of a PIM to a given PSM).

The ODAC Guideline for PIM defines UML profiles related to the different specifications. These guidelines contain profiles and corresponding rules for their application (information profile, calculation profile). In addition, the ODAC project also refers to the EDOC profiles of OMG [72].

In the ODAC project, however, there are no statements about concrete tools for the development environment or about transformation rules required for automated model transformation or code generation. ODAC describes a development process and the results of the process steps as well as the type of results. In addition, ODAC does not define roles with regard to the persons involved and their responsibilities.

2.3.2 MASTER

The MDA-based method MASTER originates from the European IST project (Information Society Technology) of the same name. This method defines eight project phases, each of which consists of a large number of subactivities. These phases are (see Fig. 3):

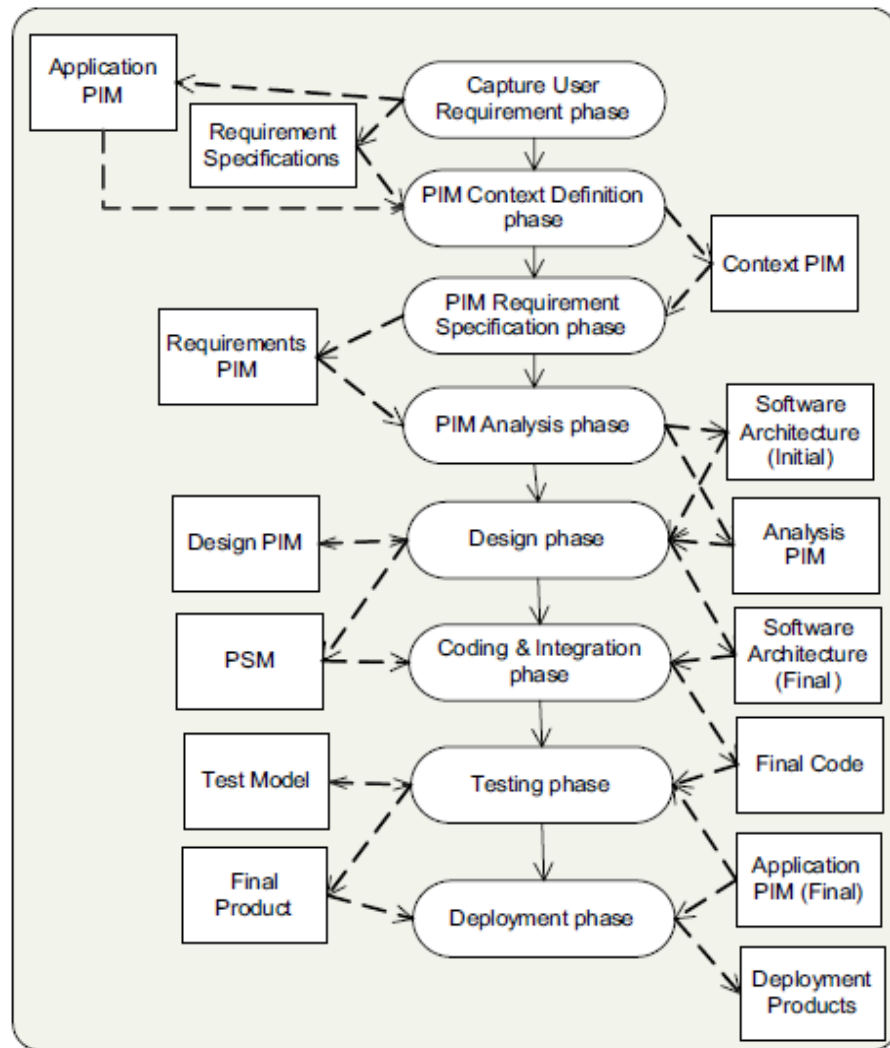


Fig. 3: The MASTER Process [7]

- **Capture User Requirements:** The aim of this phase is to identify and record customer requirements. This phase provides three results: a formal representation of customer requirements in the form of an application model, a first application PIM and a first specification of the functional requirements.
- **PIM Context Definition:** In this phase, the definition of the system is to be developed. The goals for development must also be defined. To this end, external actors are identified and the most important cases of application of the system are

described. Business objects that are exchanged between the actors and the system are also identified.

- **PIM Requirements Specification:** The main task in this phase is to refine the results of the PIM context definition. For this purpose, the use cases are specified. In addition, the non-functional requirements are defined and described as well as their relationship to the functional requirements.
- **PIM Analysis:** In this development phase, the main focus is on describing the system functionality and the QoS aspects (Quality of Service) of the system. The results of the previous phase are taken into account.
- **Design:** In this phase of the development process, all requirements can be transformed into a platform-independent design and modeled as PIM. Based on this design, the refinement can then be created as a platform-specific design and a PSM.
- **Coding and Integration:** After the MDA approach, the source code is automatically generated on the basis of the PSM using code generators according to the defined transformation rules.
- **Testing:** Test cases will also be created automatically based on a model by generators. This test model represents a further refinement of the PIM.
- **Deployment:** This phase describes the delivery of the system to the customer.

The method consists of a very rigid sequential process and describes the activities required for the application of MDA technology. It focuses on the process of capturing functional and non-functional requirements and on the system boundary. However,

important questions remain unanswered. During the coding and integration phase, it is specified that the source code is generated automatically. But when, how and on what basis the transformation rules are created remains open. The same applies to the creation of test cases. In [64] Larrucea et al. note that the agile aspect, especially the agile modelling, is increasing and see it as the focus of their further work.

2.3.3 DREAM

DREAM (Dramatically Effective Application development Methodology) is a combination of Product Line Engineering (PLE) and the concepts of the model transformation in MDA. Fig. 4 shows the stages of the process.

The particular phases of this process include the following activities:

- **Domain Analysis:** This phase collects and describes the common system requirements of different organizations within a domain. In addition, the differences between organizations can be identified.
- **Product Line Scoping:** In this phase, the product line (cf. platform) and the target environment are defined.
- **Framework Modelling:** A PIM defines the general architecture for the intended members of the product family. Including the necessary relations and conditions.

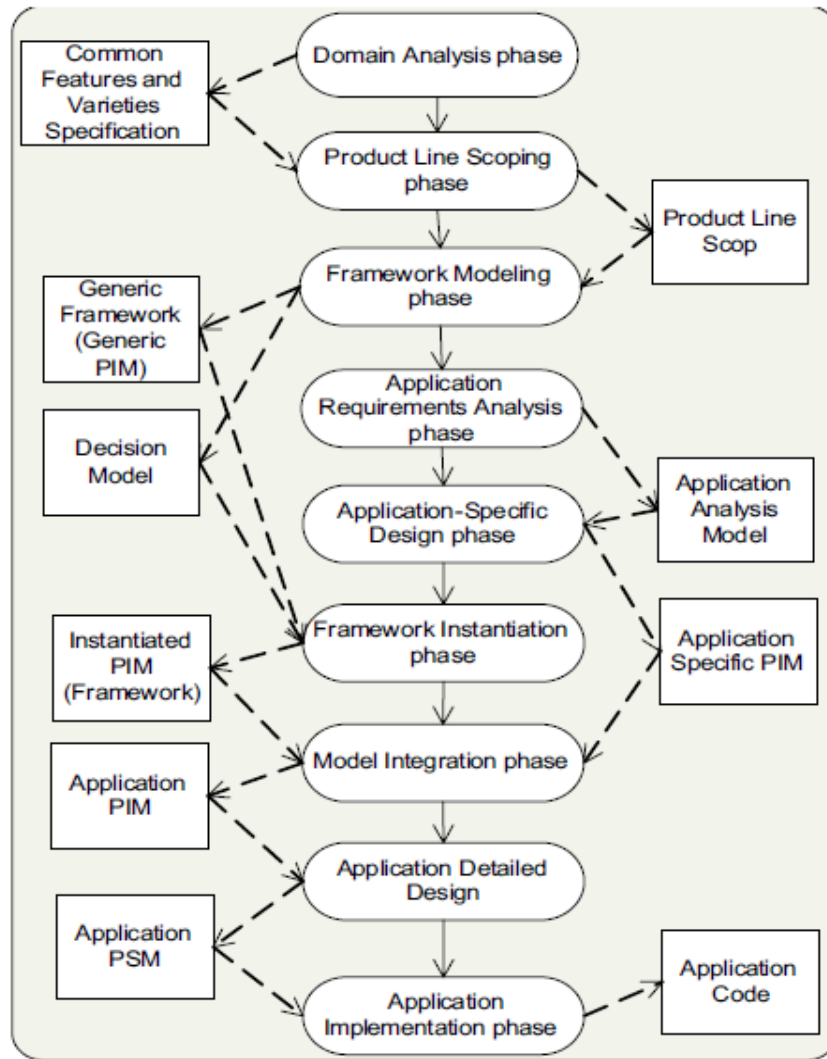


Fig. 4: The DREAM Process [7]

- **Application Requirements Analysis:** The result of this phase is the so-called Application Analysis Model and describes the functional requirements and features of the application.
- **Application-Specific Design:** Here, the application analysis model is refined and a platform-independent design model is created. This result is referred to as an application-specific PIM.

- **Framework Instantiation:** In this phase, the general architecture is concretized by considering the application-specific variants. The result is the instantiated framework PIM.
- **Model Integration:** In this phase, the specific application PIM and the instantiated framework PIM will be merged into one model.
- **Application Detailed Design:** The result of the previous phase is refined based on platform-specific properties. The next result is the PSM.
- **Application Implementation:** In the final phase, the executable code and additional artefacts (such as the database) is generated from the PSM.

By focusing on the definition of the product line and establishing a common architecture for all applications of the product line, the DREAM method responds to the fact that for applications of a product line or family alone, the generative approach pays off. Stahl et al. [91] have also recognized this and described it as advantageous. Nevertheless, the DREAM process is relatively strictly sequential. References to iterative or even agile approaches are not given. However, there are starting points for this: e. g. the features identified in the Application Requirements Analysis. They are fine-granular enough and can therefore be a solid basis for a Feature List (in the agile method FDD) or a Product Backlog (in Scrum).

2.4 Critical View / Problems in MDD Projects

2.4.1 Effort/Cost Drivers in MDD Projects

One of the basic problem areas of MDD is the high initial expenditure of an MDD project. This is due to the provision of the infrastructure and the development of the necessary DSLs, metamodels and transformations for the target platform. As already described, this is called "MDD infrastructure" or "Domain Architecture" [91]. The development of these artefacts is described in all the methods studied. The elements of the Domain Architecture were illustrated in Fig. 1 in chapter 2.2.3.

The development of the Domain Architecture with the three parts "domain", "transformations" and "platform" is an expensive and time-consuming process at the beginning of the MDD project. Fig. 5 below illustrates the two general phases of a typical MDD project and shows this effect. Phase I represents the first construction phase of the MDD project with the development of the Domain Architecture. In phase II, the actual application development with the implementation of the business logic takes place. Here the DSL, generators etc. are used.

By speeding up the initial phase and the development of the Domain Architecture (phase I) and an earlier start of the development phase (phase II), an important point of criticism could be defused, namely the high initial costs. An earlier return on investment and, if necessary, lower overall costs for the project can be achieved. This thesis also examines the extent to which this can be achieved with agile methods and techniques.

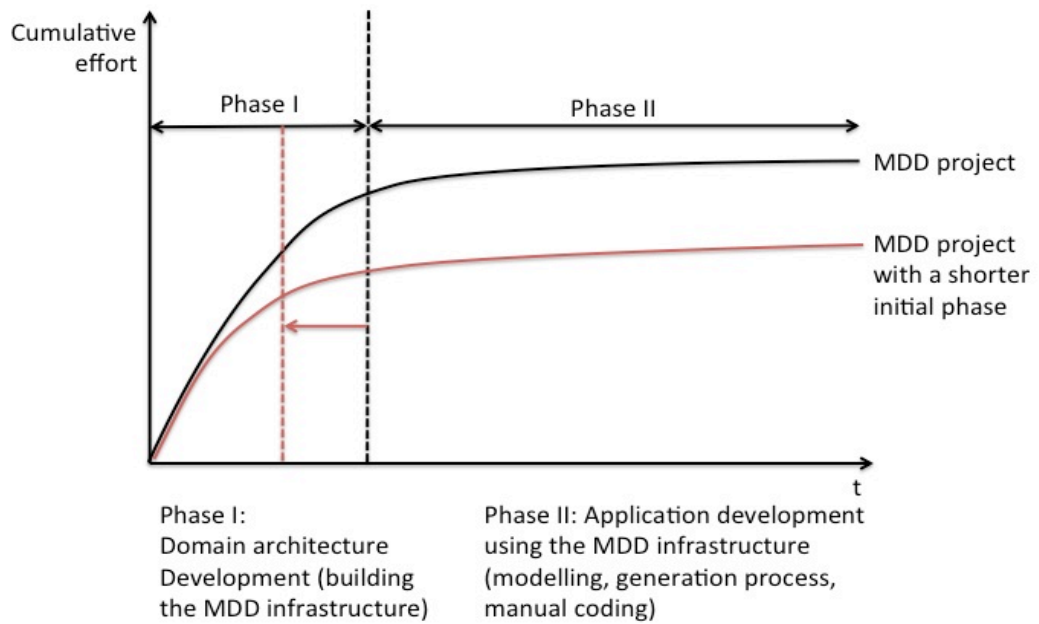


Fig. 5: The two Phases of an MDD Project

2.4.2 Limitations of Model-Driven Development

As learned, several studies documents that the promised benefits of the model-driven development can be achieved. However, model-driven development involves some problems and risks. In [47], Hailpern and Tarr name the following problems that occur in the context of MDD:

- **Redundancy:** According to the authors, e.g. the concept of different views or viewpoint models on a software system is a problem. These are “*multiple representations of artefacts inherent in a software development process, representing different views of or levels of abstraction on the same concepts*”. Additionally, there is the problem that these are “manually created, duplicate work and consistency management is required”.

- **Round-trip problems:** The authors further criticize the complex interrelationships between the different levels of abstraction. “*The more models ... are associated with any given software system, the more relationships will exist among those models.*” Changing an interrelated artefact will affect one or more of the other artefacts. “*The worst forms of the round-trip problem generally occur, when changes occur in artefacts at lower levels of abstraction, such as code ...*”. The authors conclude, “*the basic problem is that the introduction of multiple, interrelated representations implies the issue of assuring their mutual consistency.*”
- **Additional complexity:** Additionally, the authors state that with a growing number of development artefacts the complexity in the relationship between them increases and also of the development tools. Hailpern and Tarr write that “*it remains to be seen if people have an easier time managing a relatively small number or large artefacts with fewer relationships, or if they manage better with a large number of more specialized artefacts, with a correspondingly greater number of relationships.*” And they conclude, “*a process may be simple the first time through, but given the complexity that has been ‘moved,’ it may be impossible (or prohibitively expensive) to maintain, debug, or change the resulting artefacts in the future.*”
- **MDD languages:** Hailpern and Tarr identify the standardization of modelling notations such as UML an important foundation for the success story of model-driven development. But the powerful extension-mechanism of the UML 2.0, the Meta Object Facility (MOF)[71], “*enables UML to be extended almost arbitrarily*”. And they criticize that “*this dearth of semantics complicates the*

correct usage of UML extensions, reduces their expressive power, and limits the ability of tool vendors to provide reliable, consistent model technologies.”

Heijstek and Chaudron also identify different effects of model-driven development on the software architecture process in [49]. Possible problems may arise from these points:

- Late changes beyond project scope have a more fundamental impact.
- Increased likelihood of scope creep due to ease of change.
- A code generator is an additional application that needs to be developed, tested, delivered and maintained in parallel.
- Mismatches between metamodel domain and client reality need to be acknowledged.
- More tooling is needed to support the MDD process.
- Architectural descriptions need to be more extensive, formal and structured.

As a result of their study in a large-scale industrial software development project, Heijstek and Chaudron conclude that *“more effort should be planned up-front so that all requirements that impact the metamodel are known upfront and so that architectural design documentation is of sufficient quality, detail and completeness”* [49]. The development and maintenance of metamodels and generators was perceived as a new, time-consuming activity. Due to this additional effort and the novelty of the technology in the project, there is a risk that the expected productivity gain will not be achieved.

In another paper Singh and Sood [88] discuss the perspective of model-driven development. They identify following additional disadvantages of the MDA-approach:

- The MDA standards still not support the full development process and they don't meet the requirements of the software industry.
- The existing lack of tool support could become an adoption barrier to MDA.
- Due to the complexity of UML there are additional skills required.

Despite these criticisms, all authors come to the conclusion that the MDA approach is *“fast becoming the latest software development approach of present and future.”* And if there will be a better methodology and tool support, *“MDA seems to have a great future ahead”* [88].

2.4.3 General Limitations and Criticisms

In their review [7], the authors Asadi and Ramsin present some of the MDA-based methods described above and compare their properties. The authors conclude that MDA does not make sense without a software methodology and the tools that implement the most important concepts and standards. In addition, the authors summarize critically:

- The examined MDD-based methodologies support software engineering activities insufficiently and crosscutting activities are not sufficiently considered in most of the reviewed methods.
- Most of the methods give no suggestion for tool usage. Also, they don't describe the relevant tool-based activities.
- The development of platform independent models (PIM) and platform specific models (PSM) is usually well supported, the computation independent model (CIM), however, is mostly ignored.

- Conventional OOA and OOD techniques are commonly used to produce platform independent models (PIM).

Chitforoush et al. also investigate different methods and their support for model-driven architecture in [21]. Similar to Asadi and Ramsin in [7], the authors come to the conclusion that in principle only very few MDA methods are available and their description is usually very incomplete and inaccurate. This was also established and confirmed in the previous observation. Based on their findings, Asadi et al. and Chitforoush et al. have developed their own approaches to describe a development process for MDA. While Asadi et al. in [8] focuses on the life cycle of system development, Chitforoush et al. describe in [21] a process framework for MDA.

2.5 Summary

In the previous chapter the relevant terms from model-driven development were defined, which are important for further work. These are in particular the terms model, view and the central concept of Domain Architecture. As there is no established process model for model-driven development in the field of commercial software development, three methods originating from the academic field were presented. These were described with their essential properties and process steps. In addition, a number of criticisms of these process models and model-driven development in general have been put together. The different authors often point out missing or incomplete development processes that support model-driven development efficiently. They also point to the additional complexity and increased effort involved in developing the necessary MDA platform (Domain Architecture).

3 Agile Development and Modelling – Existing Approaches

The model-driven development attempts to be more effective and more profitable through the industrialization and automation of the software development. According to [65] a different approach to improve productivity is the use of agile process models like Scrum [28], eXtreme Programming (XP) [13] or Feature Driven Development (FDD) [24]. Agile software development is widespread and on the rise. The large number of agile methods is a clear indication for this. The different methods have their own specific priorities, their strengths and weaknesses. In Gartner's "Hype Cycle for Application Development" agile methods emerge in 2007 for the first time and were classified as "already used and proven." Gartner estimated in 2008 that agile methods will evolve rapidly over the next few years and it will take five to ten years for agile methods to become widely accepted and widely used in companies. From today's perspective, this can be clearly confirmed.

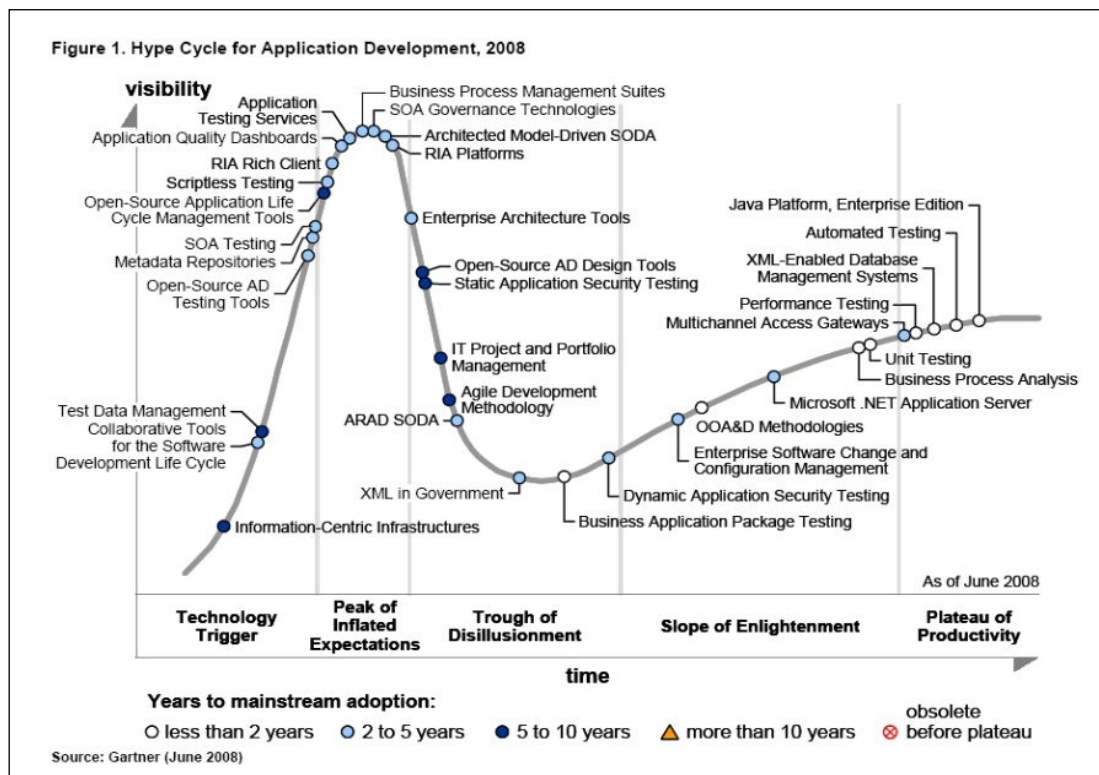


Fig. 6: Hype Cycle for Application Development, Gartner, 2008

Repeatedly emerging new agile methods also show, however, that there is no "right" agile method. Adaptation of agile practices and techniques to existing and new development processes shows the strengths and advantages of agile software development: flexibility and adaptability.

Agile process models attempt to find a compromise between a “small process” and “too much process”. The focus is not on the production of documentation. In extreme cases, this means that the source code is the only documentation. For these process models the early production of source code and the continuous delivery of executable partial results is an important aspect of the process. The communication with the end user and the direct feedback are the most important success factors for a project and facilitate quick reactions on requirement changes [35]. In agile development processes modelling often plays a subordinated role. The requirements will be documented via “user stories” (XP) or “features” (Scrum, FDD). They are summarized either in product- or sprint backlogs (Scrum) or in feature sets (FDD). This doesn’t mean that there is no documentation or modelling in the development process. But only FDD describes modelling as an explicit step in the development process.

3.1 The Agile Approach

In the 1980s and early 1990s there were many views on development methodology and how to write better software: Risk management, careful project planning, formalized quality assurance processes and the careful use of analysis and design methods and CASE tools as well as controlled and monitored software development processes.

This view came from the software engineering community that was involved in building large, long-lasting systems. Large teams working for different companies had to develop software together. Frequently distributed teams often needed a long time to develop the software. This often resulted in considerable expenditure for planning, construction and documentation.

Transferred to smaller projects, these practices and formalisms brought with them an immense overweight of organizational effort and dominated the development process. The dissatisfaction with heavyweight development processes prompted a group of software developers in the 1990s to publish the new "agile methods". Therefore, they aim to refocus the development team's focus on the software itself instead of design and documentation.

Since then, some agile methods have proven successful in practice. The most common methods are eXtreme Programming [13], Scrum [28][85][86], Crystal [25][26], Adaptive Software Development [50], DSDM [92][93] as well as Feature Driven Development [6][24][73].

The success of these methods has also led to an integration of traditional methods with agile development approaches. Examples of this are Agile Modelling [4] as well as agile instances of the Rational Unified Process. A comparison of agile software development methods is also found in a study of Pentasys AG [78].

3.1.1 Agile Principles

The agile manifesto [2] consists of four key messages, behind which are a further twelve principles of agile software development [3]. Sommerville [89] summarizes these key messages and principles together as follows:

- **Customer involvement:** Customers should be closely involved throughout the development process. Their role is to provide and prioritize new system requirements and to evaluate the iterations of the system.
- **Incremental delivery:** The software is developed in increments with the customer specifying the requirements to be included in each increment.
- **People not process:** The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
- **Embrace change:** Expect the system requirements to change and so design the system to accommodate these changes.
- **Maintain simplicity:** Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

This summary of Sommerville shows the difference to the conventional, established and often heavy-weight software development processes: namely, the involvement of the

customer in an easiest possible and transparent development process, the delivery of intermediate results for verification and early use.

3.1.2 Agility

The term "agility" is generally understood as mobility, adaptability and flexibility. In the context of software development, this relates to the agile manifesto based on the following factors [2]:

- The early provision of functioning software.
- Daily collaboration and personal communication between all those involved.
- The willingness and ability to always accept new customer requirements and to take them into account.
- The team organizes itself and achieved efficiency gains.

This results in a high degree of adaptability of the development process. This is an important part of agile software development.

3.1.3 Agile Techniques and Practices

Based on the agile manifesto and the underlying principles, various development processes like Scrum, FDD, etc. emerged. Within these so-called light-weighted development processes emerged various agile techniques and practices. The following table maps known agile practices to the disciplines of a software development process.

Discipline	Agile practise / technique
Team building	Dynamic teams
Estimating and planning	Story points
	Ideal time
Iteration	Retrospectives
	End game
	Milestones first
	Early incremental planning
Requirements	Consume your own output
	Community involvement
	New and noteworthy
Modelling	Assume simplicity / simple design
	Architecture envisioning
	Model storming
	Just barely good enough
	Iteration modelling
	Multiple models
	Document continuously
Implementation	Embrace change
Implementation (cont.)	Enabling the next effort is your secondary goal
	Incremental change
	Multiple models
	Pair Programming
	Refactoring

	Collective code-ownership
	Programming conventions
Build management	Continuous integration
	Small-sized releases
	Live betas
	Build to last / build for change
Test	Test first
	Continuous testing
	Test-driven design

Fig. 7: Agile Techniques and Practices

The listed techniques are also frequently used independently of specific agile process models and combined with existing development processes. They are therefore very interesting as part of a definition of a new development process, because they cover many disciplines of software development.

3.2 Opportunities and Limitations of Agile Approaches

Sommerville describes in [89] project types that particularly suited for agile methods. So the author named the product development of small or medium-sized products and also custom system development within an organization, “*where there is a clear commitment from the customer to become involved in the development process and where there are not a lot of external rules an regulations that affect the software.*” In the development of large and in many cases complex business applications it is common practice to use more formal process models with strong administrative aspects, such as for example the V-model. However, Eckstein describes in [34] that agile process models can also be used in

large projects instead of the heavyweight process models. But in [80] Ramesh et al. identifies some challenges for the application of agile development processes in large (especially distributed) teams. As an example there is the conflict between communication need and communication independence. Agile development processes are based on informational communication rather than detailed documentation. But in large projects with many team members there is a need for formal methods such as detailed specifications or architectural design to give the developers the information needed.

In [98] the authors indicate the importance of face-to-face communication in projects as a limitation of agile processes for distributed teams. Turk et al. also explain several limitations for agile processes. These are amongst others:

- No or limited/poor support for distributed development: The agile principles [3] give guidance on the implementation of an agile approach. However, principles such as “continuous delivery of valuable software” lead to a variety of challenges in distributed teams. Therefore, the early and continuous delivery of software requires a stronger collaboration between all locations as in non-distributed project teams. It’s the challenge not to accomplish several individual systems on the various sites but one coherent system. Furthermore, it is very difficult to achieve a close cooperation between customers and developers. In addition to the spatial distance there are often also cultural differences, and large differences in time zones can complicate the cooperation too. Nevertheless, all team members must get a common understanding of the business requirements. In [35], Eckstein describes different roles (e.g. the “traveller”) to enhance the communication and collaboration in distributed project teams.

- No process support to identify reusable software components: In agile processes the focus is on the development in short cycles and an early delivery of valuable software. This precludes the development of generalized solutions [99]. But it is clear that reusability could yield long-term benefits. According to [98] the development of reusable software components or generalized solutions is best assigned in teams that are primarily engaged in the development of reusable artefacts. Turk et al. [98] refers to a study [10] showing it's best to separate the product development from the development of reusable software components. The development of reusable software components requires a special attention to the quality, because errors in these components are often of greater relevance. In fact it is desirable to develop reusable components in a timely manner, but according to [98] it is not clear how agile methods can be adapted accordingly. Hummel and Atkinson discuss a possible solution to this problem in [53]. The authors propose to integrate the identification of reusable components tightly to the test-driven development cycles.
- Problems in refactoring large and complex software systems: Agile methods are based on the premise that good design is achieved through constant refactoring [39]. This cannot be sustained in large complex systems. The increasing dependencies between software components make the code refactoring over the entire application costly. At the same time, it increases the risk of errors. Turk et al. [98] also refers to software, whose functionality is so closely coupled and integrated that it isn't possible to develop the software incrementally. In agile projects test-driven development (TDD) is a well-proven method to reduce the risk of errors during the refactoring process. But, with the increasing complexity

and the growing number of dependencies between components the effort for the maintenance of test cases increases too.

But, according to the study from Parsons et al. [74] almost 40% of the surveyed IT professionals use one or more agile methods or techniques in software development. Close cooperation with the customer and refactoring are commonly referred to as the agile techniques with the greatest benefit in terms of quality, productivity and satisfaction.

3.3 Agile Methods for MDD Support

The model-driven software development enables a productive and quality-driven development of software systems, especially of software product lines. The existing development processes for MDD projects are, as shown in Chapter 2.3, however, incomplete, and often rigid and sequentially. Moreover, the problem exists that the development of the infrastructure of an MDD project, the so-called Domain Architecture, at the beginning of the project caused a lot of effort (see section 2.4.1). However, it is necessary for the development of the application, as it provides the necessary elements for the modelling language and the required model transformations. Agile methods are more flexible and iterative in their processes, and they focus the goal, to provide useful application parts as early as possible. Therefore they address exactly the weaknesses and limitations of the model-driven development. These are reasons to integrate agile approaches in model-driven processes and to consider how agile techniques and practices can be utilized in this environment.

Now, if an agile process should support model-driven software development, there is the fundamental question which agile techniques and existing agile process models provide

the most appropriate approaches. For this consideration is initially important to identify the agile process models that emphasize the creation of the artefacts that are necessary for the model-driven software development. Information on this gives a study of the PENTASYS AG from Germany in its Status Report 2012 [78] about the most important methods of agile software development. The study compares 25 agile process models with respect to their processes and key aspects. To support the model-driven software development, the methods are interesting, which primarily emphasise the modelling in the requirements management (RM) and the system design as well as the technical design (SD). The figure below shows the characteristics of those methods that emphasize the mentioned topics most.

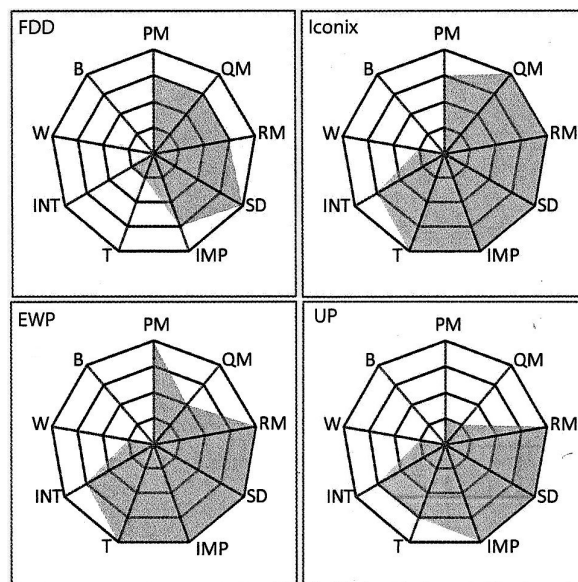


Fig. 8: Characteristics of the selected Methods (Excerpt from [78])

The methods are:

- **Feature Driven Development (FDD):** The specialty of FDD is that it emphasizes the modelling as a separate process step and provides an overall model as starting point for further development. This method is therefore presented in more detail in section 0.

- **Iconix:** Iconix represents a UML-based, lightweight software development method. The Iconix process consists of four phases, "Requirements", "Analysis / Preliminary Design", "Detailed design" and "Implementation", where the last three phases will be done iteratively until the software met the desired customer requirements. Iconix uses four UML-based diagrams (Use Case, Sequence Diagram, Domain, Class), to perform prioritized use cases iteratively into source code. In each phase a check is made of the previously completed work and, if necessary, an adjustment.
- **Eclipse Way Process (EWP):** The Eclipse Way Process is based on the way, as the widely used open source development environment Eclipse is developed. It is a combination of agile methods, methods from the open source development and working practices of large, distributed teams. The techniques of the Eclipse Way Process can be put together like building blocks and adapted to current needs. Due to the strong focus of the process on the component-oriented design and technical approach and techniques for distributed teams, the Eclipse Way Process provides valuable suggestions for an agile model-driven development process.
- **Unified Process (UP),** and the derived process models **Rational Unified Process (RUP), Open Unified Process (OUP),** and in particular the **Agile Unified Process (AUP):** The Unified Process is a popular process framework, and the Rational Unified Process the best-known manifestation [63]. The Unified Process is essentially iterative and incremental, regarding to the requirements use case driven, architecture-centred and has a strong emphasis on the early risk assessment. The very slender use cases driven approach for the requirements analysis and the architecture centred design seem to qualify the Unified Process

as a valuable source for ideas for an agile model-driven development process. The simplified version of the Unified Process, the Agile Unified Process, has also the goal to enable greater agility within the Unified Process. Scott Ambler combines the disciplines of business modelling, requirements analysis as well as design in one discipline "model", in which the created models thereby only have to satisfy the claim to be, "just good enough". AUP is designed for medium-term projects and medium sized teams.

In addition to the four presented process models must be mentioned AMDD (Agile Model-Driven Development). The properties of AMDD do not fulfil the initially mentioned criteria with emphasis on requirements management and system design (see Fig. 9), however AMDD contains interesting approaches in the context of agile modelling techniques. AAMD is described in more detail in section 3.4.1.

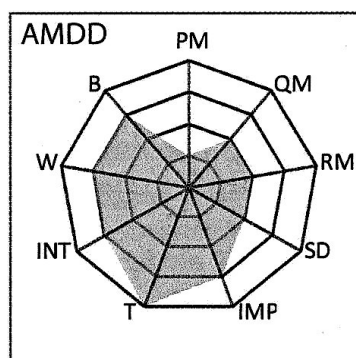


Fig. 9: Characteristics of the AMDD Methodology (Excerpt from [78])

Besides the possible suitability of an agile methodology for model-driven software development it is additionally relevant, how well known and widespread the respective agile method is. A possible agile method for MDD will be more accepted, if it is in the style of a in practice widespread agile methodology. Regarding the awareness level of the

previously considered agile methods and their distribution the PENTASYS study [78] can be used, too. The study uses search results on Google, and the number of books at Amazon to get an indication at spread and popularity. By combining the search results of Google and Amazon was formed a normalized "relevance index". That is, the method with the highest combined score gets the relevance index 1. Fig. 10 at the next page shows the result. As can be seen, however, none of the previously mentioned process models have a high degree of popularity or widespread. Here still XP, Scrum and TDD are the dominant development models. For a definition of a process model for the agile model-driven development, this should be taken into account. It should be considered agile techniques also from these popular methods, whether they are suitable for this purpose.

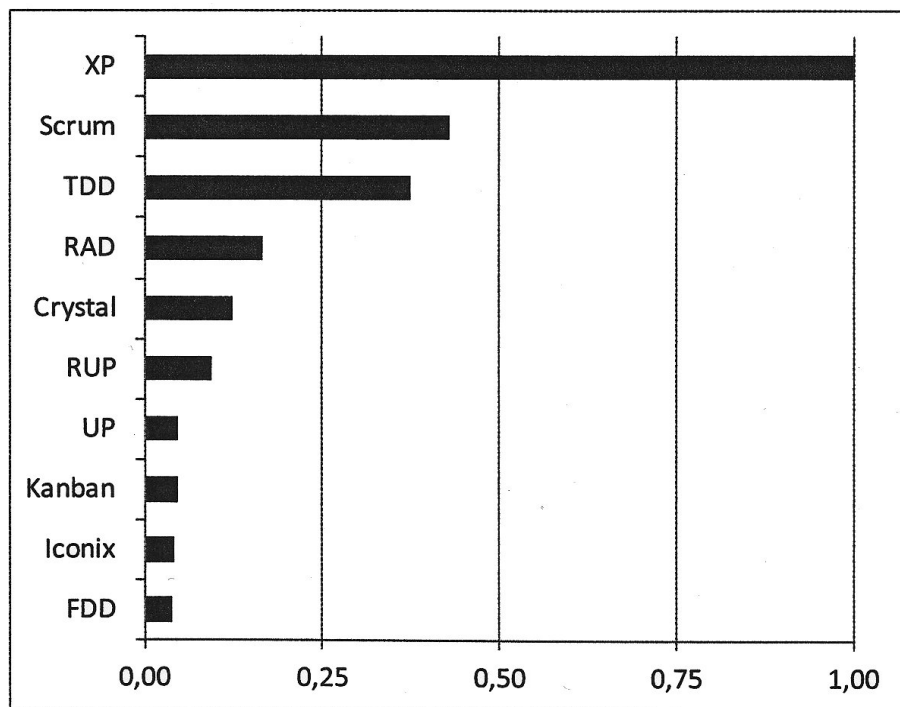


Fig. 10: Awareness Level and Distribution of Agile Methodologies (see [78])

3.4 Existing Approaches, Solved Problems and Limitations

3.4.1 Agile Model Driven Development (AMDD)

One approach to combine MDD with agile techniques was presented by Ambler [5] with AMDD (Agile Model Driven Development). The difference to traditional MDD is in the draft of models. In MDD, first extensive models are created before starting to write the source code, in AMDD the aim is the creation of models with a minimum of effort (e.g. on a whiteboard).

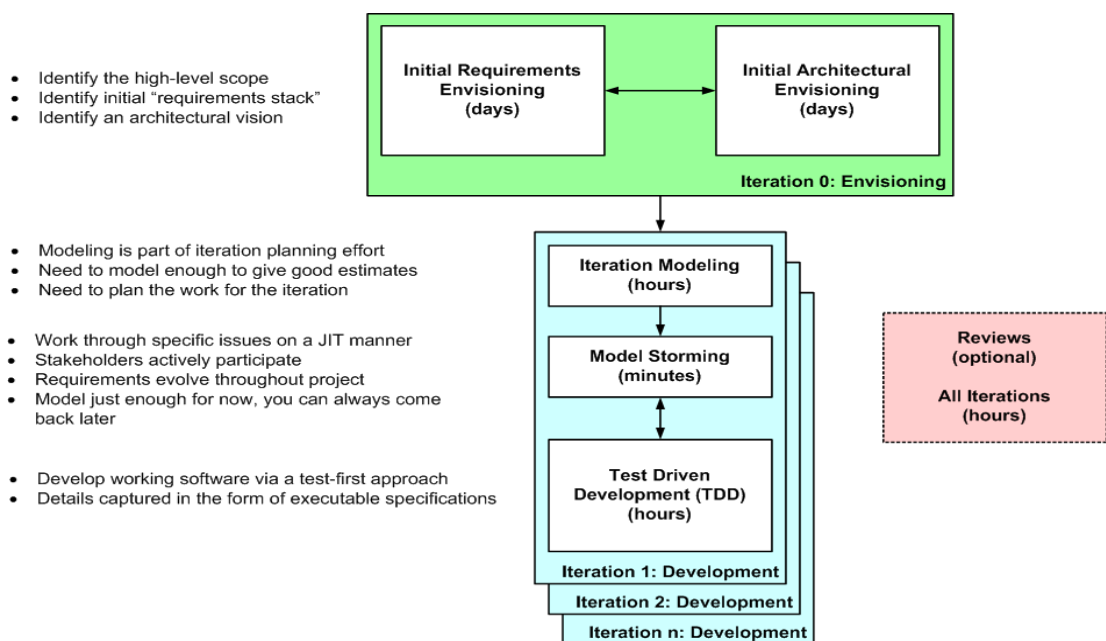


Fig. 11: The AMDD Process [5]

The motivation for this is to initially reflect only the most important basic requirements. The models should be "just good enough" for the current workload. In further iterations (iterative development), the requirements are refined and optimized.

The AMDD process includes the following phases:

- **Envisioning:** At the beginning of the project there is closely work with stakeholders to identify the most important requirements and to model scope of the system. The system architecture is also roughly modelled to specify the technical direction. The entire model is in this phase, relatively little detailed and just enough. The important thing is that the problem is to be understood.

- **Development Iterations:** At the beginning of each iteration, the team plans its work and prioritize the requirements. Through the close cooperation between stakeholders and developers in each iteration, new or expanded requirements are developed. The development is to take place to Ambler through test driven development (TDD).
 - **Initial Requirements Modelling:** The objective of this phase is to develop a good idea of the project. These include a first usage model, domain model and user interface model.
 - **Initial Architecture Modelling:** In this phase, a first architectural model is developed. This determined the technical direction of the project and has also been the starting point for the project organization.
 - **Iteration Modelling:** Here, the decision on the size of the work packages is taken to be retrieved from the requirements storage. The packages in the requirements storage are typically prioritized.
 - **Model Storming / Just In Time (JIT) Modelling:** It is permissible to hold a so-called "Model Storming" if required, in which a team member gets one or

two colleagues to help in order to make a spontaneous modelling decision with more certainty. The Model Storming should take less than 30 minutes.

- **Test Driven Development (TDD):** The modelling is followed in each case directly by the coding. This is handled as in the Test Driven Development [12].
- **Reviews:** Classical reviews or code inspections are not usually carried out. Except in large teams or in large projects.

- **Release:** Within this phase final tests and acceptance tests are done to verify the functionality of the entire system. When errors occur, they will be corrected.

- **Production:** Goal here is to get the system up and assist users in using the system. The phase ends when a system or the support for the system expires.

A closer look at this description of the development process of AMDD shows, that in this case AMDD should less referred as model-driven development, but as model-based development (see chapter 2.2.2). Nevertheless, the process contains helpful hints and tips, how can be dealt with the topic of agile modelling.

3.4.2 MIDAS Framework

In their comparison of different MDA-based methods, both Chitforoush et al. [21] as well Parviainen et al. [75] mentioned the MIDAS framework. MIDAS should support the agile development of Web Information Systems. For this it uses UML as modelling language for the creation of the necessary PIMs and PSMs. In addition, MIDAS defines mapping rules for the transformation of models from PIM to PIM, PIM to PSM and PSM to PSM. But unlike the others, already presented MDD methods defines MIDAS no concrete development process. Instead, MIDAS focuses on three viewpoints, those are iteratively and incrementally to model. They describe the content, presentation as well as structure and behaviour of the application (see Fig. 12).

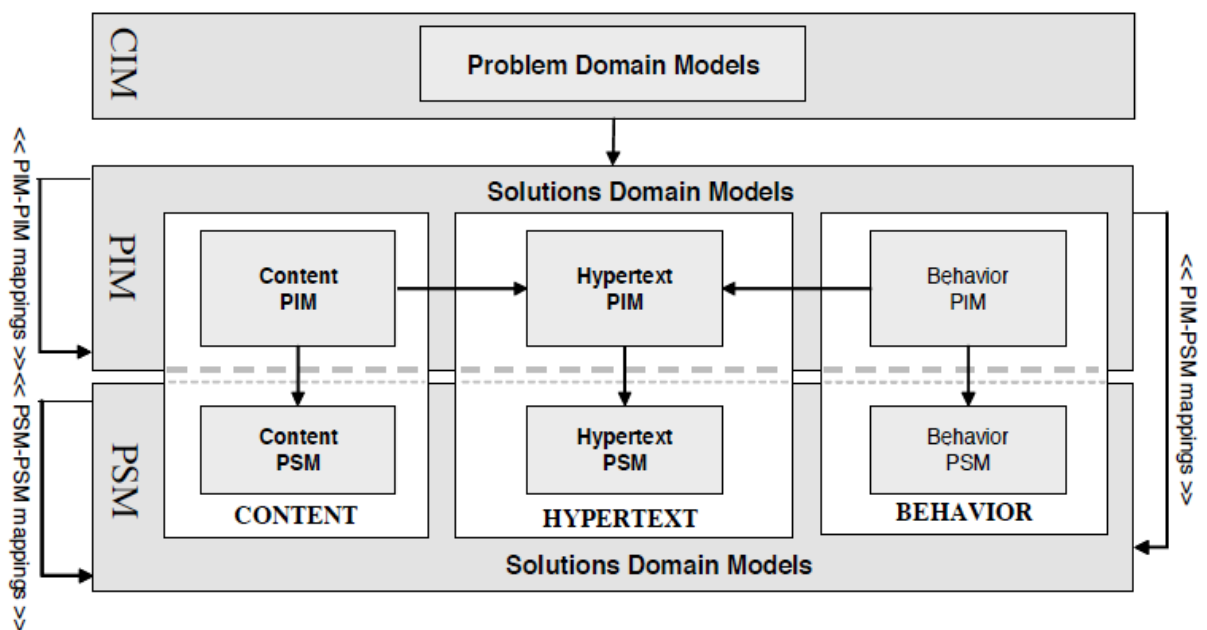


Fig. 12: Model Driven Architecture of MIDAS [20]

In another paper [20] describe Caceres et al. the experiences they have had in a case study with the integration of agile practices and activities from XP in MIDAS. According to the authors, it turned out to be positive, to develop the CIM (Computation Independent

Model) as an early general vision of the future application. The business and domain models show the relationships and facilitate the entry of new team members. The team also felt it as positive that the modelling supports the development of a common vocabulary from the beginning. As further advantages are mentioned in the article:

- The distinction of the models into three viewpoints: This was a great help in the prioritization and project planning of user stories in the first iteration.
- The use of development standards such as UML. This supports the communication between the team members.
- Development in Pair: Because of this technique, the developers felt secure in their decisions and show more responsibility in the development process.
- Continuous Integration: By the frequent delivery of software over the Web a high, well-balanced level of information in the project could be achieved. The developers described this as very important. In addition, it supported them in the task of testing software modules developed by other pairs.

Based on their case study, the authors conclude that it is important to identify the strengths of agile modelling, to guide developers in creating the models, and to make a breakdown of the different aspects (via viewpoints).

3.4.3 Feature Driven Development (FDD)

Feature Driven Development was first presented in [24]. Peter Coad et al. explain in their book FDD as a lean method for software development. The method provides the notion of "feature" in the centre of development. A "feature" is defined as a property of an application that is useful in the eyes of the customer and therefore it is an added value. Between the fine-grained functions of the features of a complete system often also exist dependencies. Therefore related features are grouped in so-called feature sets. The feature sets are also grouped according to functional criteria into higher-level groups, the major feature sets. Functional specifications comparable to the features and features sets can be found in similar form in other agile process models (e.g. the Product Backlog in Scrum).

Besides defining features, FDD provides a role model for key roles (e.g. project manager, chief architect, chief developer, domain expert), supporting roles (e.g. domain manager, release manager, build engineer) and additional roles (tester, technical writers). Typically, the team members will assume multiple roles.

Unlike other agile process models in Feature-Driven Development modelling is a defined activity in the process model. So already in the first process step, an overall model is created (see Fig. 13). The aim of this first step in the process is to get a common understanding of the content and scope of the system under development. Here, small groups of experts and developers define the functionality under the direction of one or more chief architects. Also plays the knowledge of the chief architects of the nature and use of the final product a major role because the overall model should be sustainable for all the features.

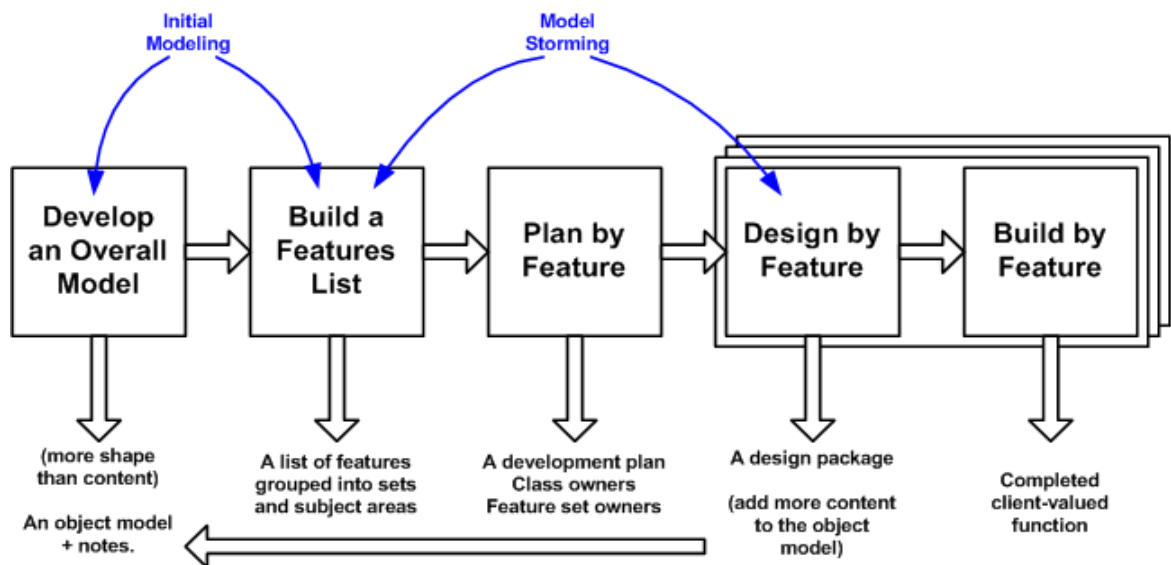


Fig. 13: Modelling within the FDD Process [6]

Another major step in the process flow, which is supported by modelling, is the design of a feature. During a walk-through of the chief programmer is developing along with the feature team a refined model. The design of the feature set and checked during an inspection before it is implemented in the next step.

Feature Driven Development is based on various best practices from the field of software development. In the field of modelling FDD provides the use of "Domain Object Modelling". Domain Object Modelling allows you to gain a good overview of the problem area as a whole. The domain object model covers only business objects that are persistent generally. Within this model graphical user interfaces and control objects doesn't matter, since this would complicate the view of the business object model. Moreover, Scott Ambler recommends in his essay on FDD and Agile Modelling [6] the agile technique of "Model Storming" for the recording and modelling of functional requirements.

With a focus on the business object model in the FDD process is achieved an abstraction and concentration on the essential relationships. The resulting model can certainly be compared with the Platform Independent Model (PIM) in the model-driven development. With the additional knowledge of the target architecture it is possible to describe a transformation of this model into the corresponding source code and thus achieve automation in terms of MDD. Any necessary additional information or model refinements can be added to the model within the process step "Design by Feature". The result is an annotated PIM, or a Platform Specific Model (PSM). Feature Driven Development can thus provide information and foundations for a possible process model for an agile approach to model-driven software development. Additional hints are the defined roles for team members as well as the best practices mentioned.

However, feature-driven development does not describe a process step for defining the architecture. Nevertheless, it is pointed out that the knowledge of the context of the application is important for developing the overall model and the chief architect should know the nature and use of the final product. But the need of the definition of application architecture is not explicitly described. However, this should be the latest on the design of the individual features so it can be considered. In case of using FDD as part of model-driven software development, the definition of the architecture is, however, a necessary mandatory step.

3.5 Consequences

For the definition of Domain Architecture as the basic framework for model-driven development, the definition of the application architecture of the future system is of central importance. At the same time, however, the modelling language must give the developers the opportunity to determine the design of the application due to the business requirements. These two perspectives have to be considered especially in an agile model-driven process. For this reason, special attention must be paid to understanding architecture in an agile environment.

In addition to the special consideration of the architecture as an essential element of model-driven projects, it is also necessary to consider the further results and artefacts of such a project. Unless elements of agile process models and individual agile practices should be applied in an agile model-driven development process, then it must be considered in what phases these techniques should be applied. It must be considered also, which results are created in these phases, and how these partial results are interdependent and influence each other. If this is right, it must be considered, in what phases a particular agile technique should be applied.

At the same time, the aims of the Agile Manifesto should be considered.

- The early and regular delivery of functioning partial results.
- Working closely with the customer for accommodating the requirements and validating the results.
- The constant readiness for change (functional and technical).

As part of the implementation of MDD projects there are two main phases. In the first phase, the Domain Architecture is defined as described in Section 2.2.3. I.e. in this phase, the necessary tools, transformation rules and DSL language elements are provided those allow developers in the following phase, to create the software using these techniques. Fig. 14 shows the results of these phases and their dependencies, and the effects of functional and technical requirements.

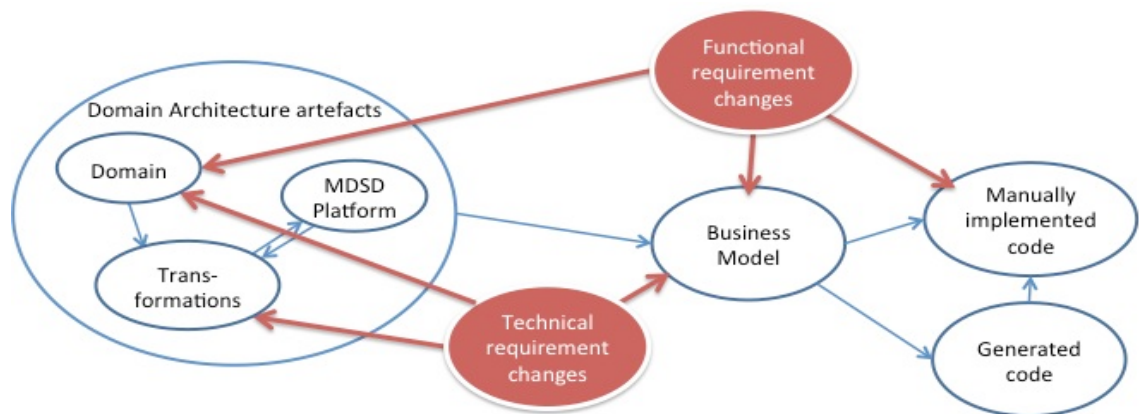


Fig. 14: Dependencies between Artefacts in MDD Projects

In the context of agile development with short iteration cycles, and the desire to provide working software early on, these two stages need not be performed in sequence but in parallel and closely integrated. Because of the dependencies of the individual results it will be particularly important to keep the impact of changes in mind. On changes to functional requirements it is possible to react quickly by adapting the business models. In the agile environment, this represents not a special case. Greater challenges are changes to non-functional requirements, which affect the elements of the Domain Architecture. Changes at the Domain Architecture can directly influence all elements of application development. This applies both to the MDS platform as well as the transformation rules of the generator, however, in particular for the domain specific modelling language. The

need for changes to the DSL may also be caused by newly discovered business requirements. To cover these requirements, new model elements may extend the DSL, with the goal to describe the requirement effectively through the modelling language. An essential task for the definition of an agile model-driven process will therefore be to take into account the dependencies between the elements of these two phases of development properly and to find an effective integration of these phases.

3.6 Summary

With AMDD [4][5], FDD [6][24][73] and MIDAS [20][21][75] three process models have been introduced that provide possible approaches for agile model-driven development. Supplemented with other agile practices of the methods identified in Section 3.3, this represents a collection of agile techniques, which provide useful starting points for the definition of an agile process model for MDD.

The definition and agile understanding of architecture and design is for the support of MDD projects also important. Ultimately the defined target architecture determines the necessary transformation of models into software. The properly chosen cut of the architecture (tailoring) and the anchoring of the design in the development process are the essential elements for a future development methodology. Another challenge for an agile, iterative and incremental development process represents the dependencies between the individual partial results. Here in the model-driven projects, the additional dependencies exist between the components of the Domain Architecture and the results of the application development.

4 Case Studies – Practical Experience in MDD Projects

Fundamental criticism of model-driven development often arises from skepticism about modeling using UML. Brambelli et al. Summarizes in [17] that the UML is generally considered too extensive, too cumbersome and incoherent and therefore cannot be used for DSLs. For this, the authors make some discourses, such as [14], about the advantages and disadvantages of the UML. In practice, this can be explained by the fact that many developers only know and use specific diagram forms of the UML. This was examined in [81] and the authors of the study also showed that there is also a difference between academic users of the UML and software developers in industry, the latter using the UML much less than the academics. Thus, it is not surprising that the model-driven development based on the UML is only very poor in practice. The complexity of the modeling language in connection with the effort of the DSL definition seems to act as a deterrent here. In contrast, text-based DSLs are also much more common in practice (see also case study 3 (4.3)).

But the authors Brambelli et al. in [17] conclude, however, that UML is still the reference language for the modeling of software systems and will continue to be the industry standard. The criticism also leads to the fact that the OMG regularly revises the specification of the UML and tries to achieve a simplification and a better manageability.

The following case studies will show some opportunities and risks in specific MDD projects from practice. The first case study is originated on a project with an insurance company in Hanover, Germany. The second case study describes the experience in an MDD project within the software product development in a medium sized software company. A third case study shows the difficulties that arise with the increasing

complexity of the domain-specific language and the difficulties that result for the product development. In addition to these three case studies, further reports by other authors from industry studies are presented and explained below.

4.1 Case Study 1: Interfaces to Legacy Systems

4.1.1 Initial Situation

The insurance company had a heterogeneous environment with Java clients, J2EE application servers and various legacy applications on mainframes, which are written in COBOL or PL/1. The realignment of the application development had defined Java as the strategic platform for new applications. The maintenance of data in legacy applications should be made in future with the new Java clients. But the access to the legacy applications was implemented inconsistently and access routines and interfaces were poorly documented. So the provision of a new interface for Java applications and the additional documentation of the interface needed a time frame of 2-3 month (40-60 person days).

4.1.2 MDD Approach

The main objectives of the development team were:

- Accelerating the development of an interface.
- Standardizing the implementation and documentation of an interface.
- It is intended to establish UML as modelling language for all applications in the Java- and COBOL-environment in the company. One UML model should be the common basis for all target architectures.

Model-driven development seemed to be a promising approach to achieving these goals. Based on annotated UML diagrams the Java client as well as the interface implementation to the legacy application can be generated. Additionally, the UML diagrams should complete the documentation gap.

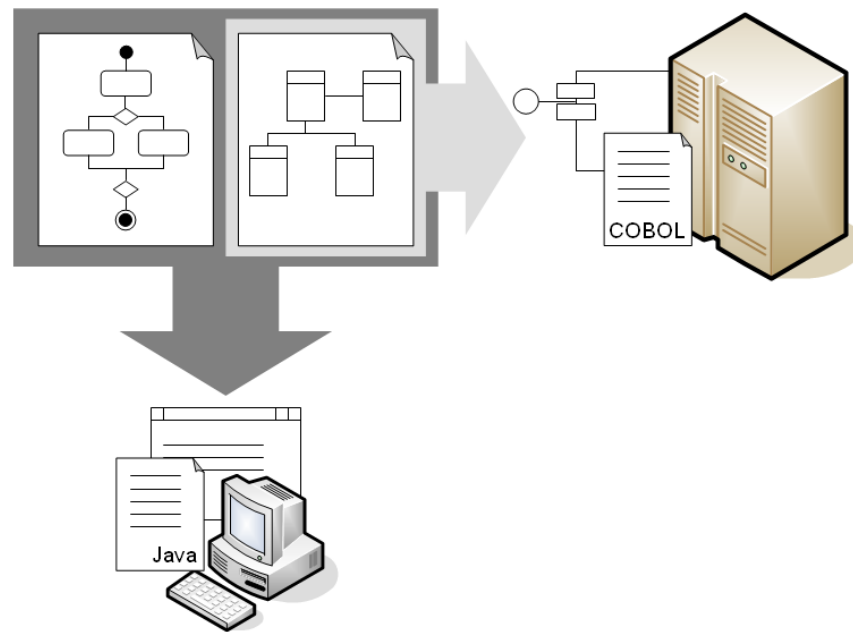


Fig. 15: Models and Generated Artefacts in Case Study 1

Based on this idea, the development team has decided to define a domain specific language as an extension of the UML class diagram as well as of the activity diagram.

- The class diagram should describe the data structure of the interface and the name of the interface methods (Fig. 16). This aimed to generate following COBOL modules:

- Access modules (for access to a database table): Here should be supported the standard requests such as Insert, Update, Delete and findByPrimaryKey as well as individual queries.
- Business modules: Due to the fact, that these modules implement individual business logic, it wasn't possible to create them directly. The aim was therefore to generate a so-called distribution module, which calls the corresponding COBOL module for every modelled function. This was intended to ensure that the provided services are also represented in the COBOL world through a standard interface.

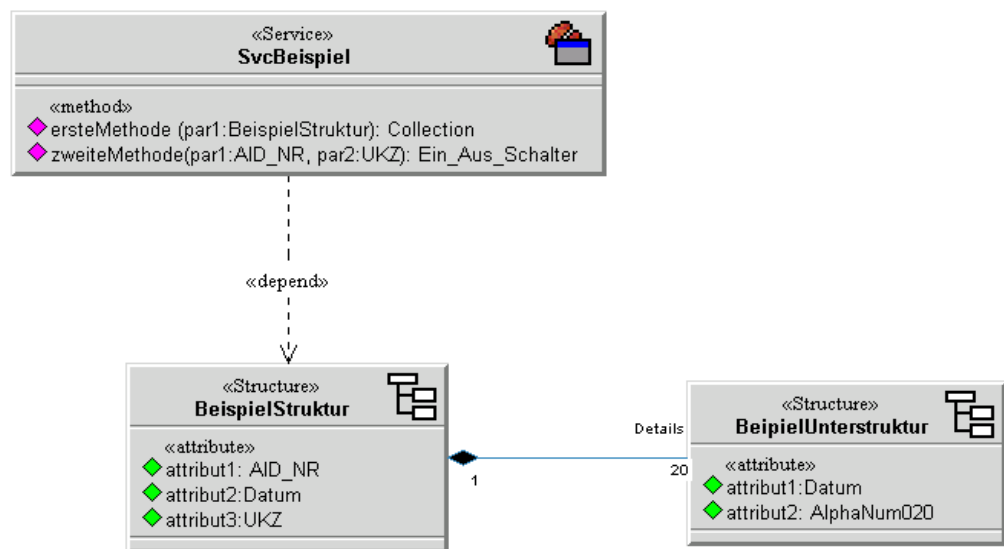


Fig. 16: Sample UML Class Diagram for the Definition of a COBOL Copybook

- Copybooks: Both, for access modules as well as for the business modules, the required data structures are generated as copybooks and integrated into the generated modules. Copybooks reflect the structure of tables, parameters or return values.

- Activity and state diagrams should give the developers the possibility to describe specialized behaviour within the client (Fig. 17). The modelled activities are intended as methods that are to be implemented by the developer individually. The states represent the various dialogs. The different transitions specify how the activity or a state is left.

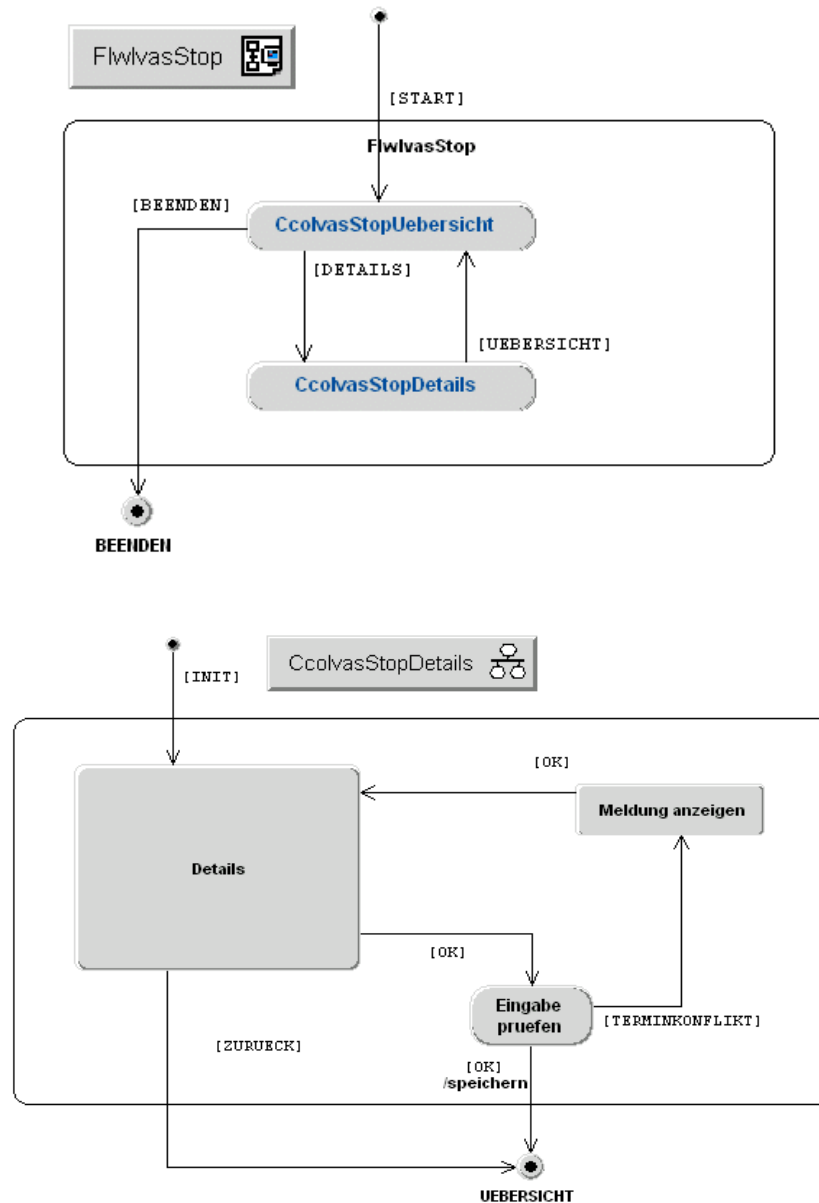


Fig. 17: Sequence in the Activity Diagram refined in the State Diagram

As a next step, the team has developed a sample Java Swing client for an exemplary interface of a legacy application. Additionally, they have defined a standardized COBOL access routine on the existing implementation. Both, the sample client and the COBOL routine were reference implementations for the development of the generator templates. In an iterative process, the team developed the generator templates and extended the metamodel of the domain-specific language as required.

In a last step, the modelling and coding guidelines for the further development were defined and the integration in the development environment was completed. Up to this point, the following effort was needed:

Activity	Effort
DSL definition / metamodel development	20 pd
Reference implementation (Java Swing client / COBOL interface)	80 pd
Template development and generator workflow	120 pd
Modelling and coding guidelines	10 pd
IDE / build management integration	30 pd
Total	260 pd

Fig. 18: Project Effort in Case Study 1

4.1.3 Result / Experience

According to a leading software architect at the insurance company, the provision of a legacy system interface for a Java application has been significantly accelerated. *“For the same things for which we have previously needed up to three months, now we need only two weeks. Consider the required effort, the return on invest is reached after only nine interfaces.”*

4.2 Case Study 2: Software Component Development

4.2.1 Initial Situation

The software company developed a J2EE-based software framework, which consists of software components that provide specific business functions for the processing of claims in an insurance company.

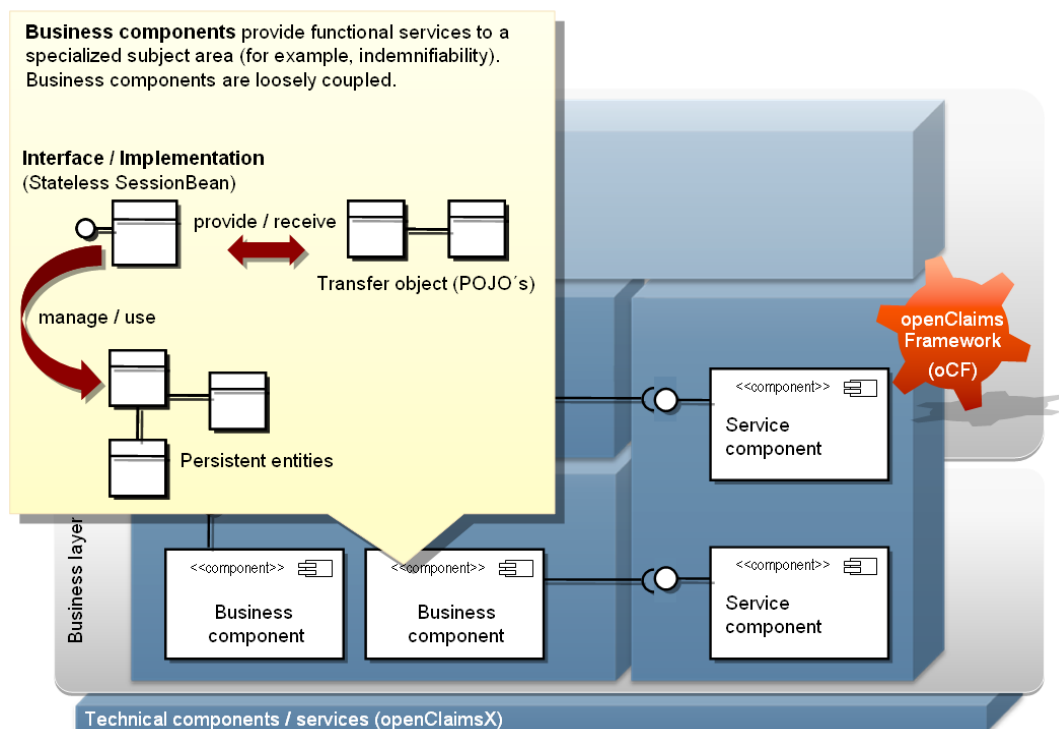


Fig. 19: Parts of the Software Framework in Case Study 2

The component interfaces were implemented as EJB Session Beans and data are delivered according to the J2EE-pattern “Transfer Object” via the interface. A proprietary framework does both, the management of persistent entities within the component as well as the management of metadata and relevant business rules.

All software components are based on the same design principle and follow this template. Therefore, the implementation of each component is very similar in large parts. Studies showed a potential for 60-70% generic source code.

4.2.2 MDD Approach

For the development of further software components for this framework, the company decided to invest into model-driven development. Based on the similar design of the components, the comparable initialization of data, similar configuration as well as the analogous transformation of persistent entities into transfer objects, it seems to be a promising approach. Objectives for the development team were to accelerate the development of new components and the easier adaptation to new architectural standards or technologies (such as EJB 3).

One of the existing software components was used as reference implementation. Based on this component, on one hand the modelling conventions (or the DSL) have been developed and on the other hand the necessary generator templates derived. The previous figure (Fig. 20) shows the developed metamodel with the definition of stereotypes and tagged values for the required model elements.

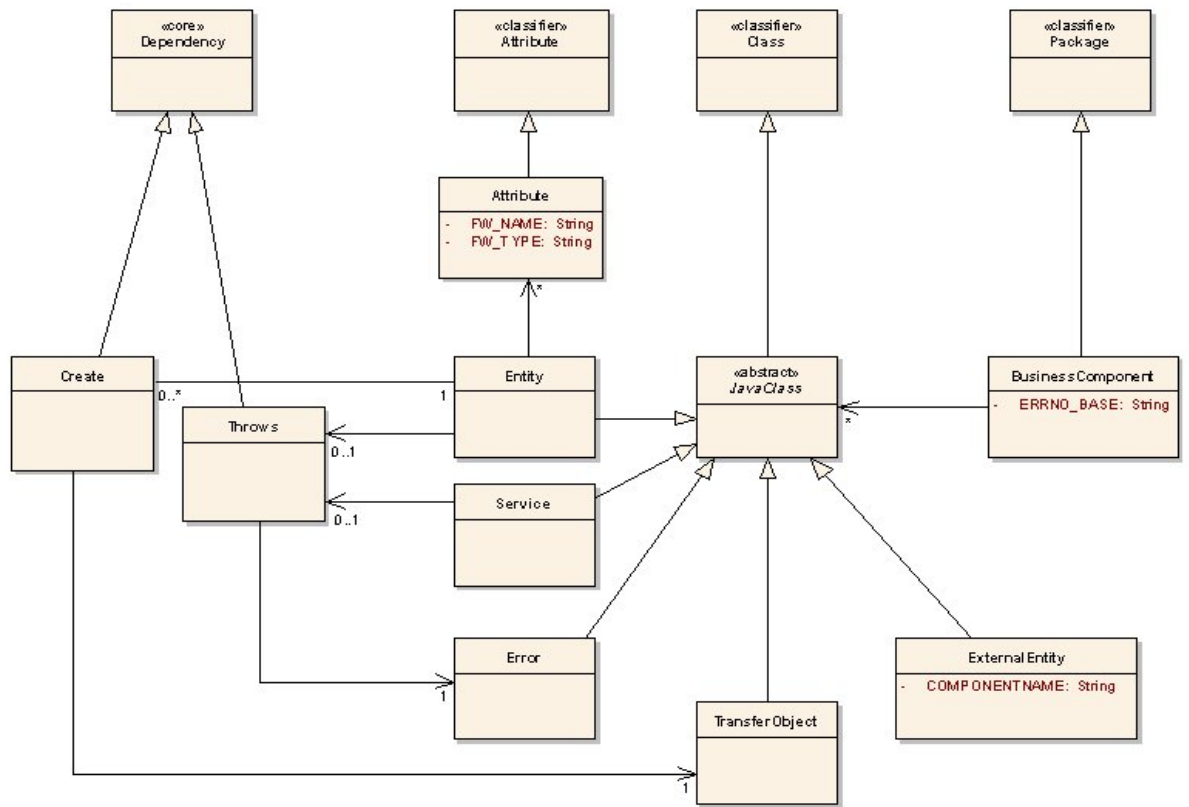


Fig. 20: Metamodel for the Description of Software Components

As a result of the generation process the following artefacts should be created:

- The interface definition and implementation of the component (as a Session Bean).
- The persistent entities according to the underlying persistence framework.
- The necessary initialisation of the component by the defined metadata.
- The necessary classes and methods for the transformation of persistent entities to the required transfer objects for the interface (and vice versa).
- Required deployment descriptors for several application servers.
- Test classes and test drivers for the component as well as different helper classes.

To achieve these results the following effort was needed:

Activity	Effort
DSL definition / metamodel development	30 pd
Reference implementation (not necessary)	0 pd
Template development and generator workflow	220 pd
Modelling and coding guidelines	30 pd
IDE / build management integration	40 pd
Total	320 pd

Fig. 21: Project Effort in Case Study 2

The development was terminated in 2006, because of new components were needed faster for a project and there was no more time to wait for the completion of the MDD environment. So, the necessary components were developed manually. The MDD environment was never completed.

4.2.3 Result / Experience

In hindsight, the termination of the development of the MDD environment has the following reasons:

- The decision to develop an MDD environment came too late. There were already too many ready-made components. The prospect of payback purely in relation to the development of new components was too low.
- Too many requirements / no iterative process: It was developed too much at once. In the beginning, too many artefacts should be generated.

- Few intermediate results that could be used.
- The management did not trust the new technology.

4.3 Case Study 3: Insurance Programming Language

4.3.1 Initial Situation

A software company with more than 1000 employees is developing software for life insurers. The typical application is a distributed Java Enterprise Application with a Web-based client, a Middle tier with clustered application servers, and several relational databases in the backend. Added to this is the need to easily integrate host-based systems. The necessary application architecture is complex. And the functional requirements too. The development of the application spread to several locations in Germany and Slovakia. The development of the architecture took place in Cologne, the functional specification in Stuttgart and the development in Stuttgart and Bratislava.

4.3.2 MDD Approach

A modeling framework has been created to ensure a uniform architecture for the whole application. This framework includes a graphically based domain-specific language on the basis of UML, which describes the microflows of the application (that is, the control of the dialog processes and transactions) as well as the macroflow over the individual application parts. In addition, a text-based DSL was developed, through which data structures and interfaces were described. Separate code structures and mixin classes (according to the Mixin design pattern) were generated from both domain-specific languages. Both languages, the graphic and the text-based DSL, were oriented on the

technology stack and both describe technical issues. The aim was to simplify the handling of technology by means of abstraction.

4.3.3 Result / Experience

With the focus on technical abstraction by the DSL, a high dependency on the selected application architecture was generated. Changes to applied technologies or the adaptation of structures within the application architecture often led to changes in the language range of the DSL. This then requires corresponding model adjustments. The productivity of the application development was thereby severely impaired. Each release of a new version of the modeling framework had immense impact and resulted in correspondingly high adaptation costs. Not infrequently, a new release meant a project stop of up to two weeks. An analysis of the situation showed that it would have been better to focus the DSL on the professional statements of the application. The mapping of the technical nature to the target architecture would have to be done transparently via the model transformation. In the current situation, there are business models intermixed with technical information regarding the target architecture. The principle of Separation of Concern has not been adhered to, and therefore changes in its effects cannot be effectively limited.

4.4 Experiences from other Case Studies

In addition to the presented case studies with own practical experience, reports and case studies on MDD projects from other companies will now be considered in the following. These cover a variety of business sectors and therefore enable a broad overview. Representing this are the case studies at IBM [19][22], ABB Robotics and Ericsson [94],

at Autoliv, Sectra and Saab Aerospace [36] and Motorola [9] called and described briefly below.

4.4.1 ABB Robotics and Ericsson

The article by Staron [94] describes the case studies at ABB and Ericsson. ABB focuses on the development of mechatronic systems with embedded software. The embedding of legacy code and the support of different programming languages were the primary focus for the support of the software development by MDD. MDD was therefore considered, because the developers expected a better portability, more accuracy and an earlier assessment of the quality of the software. On examination, the question stood on the kind of the models, the availability of the tools and the knowledge of the development team in the foreground. Ultimately, however, the high startup costs led ABB decide against the use of MDD.

Ericsson is in the business of mobile telecommunications. The department, which was involved in the case study, is engaged in the development of services for mobile platforms. Ericsson expected that MDD would improve its competitiveness by increasing developer productivity. In addition, for Ericsson reasons such as increasing quality and improving team communications were relevant. Ericsson was in favour of the use of MDD. As a modelling language UML 2.0 with custom profiles was chosen. Ericsson needed profiles because the UML 2.0 language scope of the standard was insufficient for the problem domain.

Staron draws the following conclusions from his case studies:

- Domain-specific languages should be designed and developed by the developers, who want to use MDD.
- Even excellent models do not allow complete code generation. It is not possible to dispense on manual coding.
- The MDD technology is not so far that a model-only approach like described by Brown in [18] is possible.
- The companies are struggling with the paradigm shift from the current state of software development towards MDD and rather use proven technology instead of UML model-driven process.
- The high implementation costs of MDD can adversely affect a decision in favour of MDD.

Hence Staron also draws the conclusion that, among other things, the development process is a crucial factor for the successful introduction of model-driven software development.

4.4.2 Autoliv, Sectra und Saab Aerospace

The article by Elmqvist and Nadjim-Tehrani [36] focuses on three case studies in which high reliability and security of the generated code is required. The tools used by the three companies are presented and evaluated, and then the success of the use of MDD in the companies is pointed out.

Autoliv, a German supplier of automotive components used MDD in the development of a new airbag. The main requirement on MDD was the rapid development of the software and the required low code size. Through the use of MDD was the time that was spent in the development, can be shortened by 60% compared to the hand-written code. But the generated code had to be completed manually.

Sectra is a Swedish manufacturer of secure communication systems. The main requirement for Sectra on MDD was the integration of legacy code, platform independence and security of the application. Since none of the study relied MDD tools was sufficient, a separate code generator has been developed.

At Saab Aerospace especially safety requirements for the code, and the traceability of changes to the original system were regarded as important. The specification of the systems takes place partly in a natural language, and in part by a particular model language. The model can be used for simulation of the finished system. A complete transformation of the models in the respective target languages could not take place; it was always a manual intervention necessary.

Elmqvist and Nadjim-Tehrani come, based on their case studies, to the following results:

- None of the available development tools for MDA software is reliable enough to fulfil the high requirements for their safety.
- In the case of Autoliv a great time saving can be achieved due to the use of MDD tools.
- Both, in case of Sectra as also at Saab Aerospace manual intervention was necessary because there was no tool support for all steps from the specification up to implementation.

4.4.3 IBM

At IBM, two case studies were conducted. The first case study [22] is about a project to implement a business performance management system (BPM) with MDD. Various aspects of BPM were divided into smaller, more workable pieces and modelled using UML 2.0. On model-to-model transformations so called intermediate models are generated that represent partial aspects of BPM. Based on these intermediate models, the actual program code is generated.

The observations of Chowdhary et al. from this case study are:

- MDD provides a platform on which may be developed quickly and flexibly.
- Manual additions of models and code are always needed.
- It was not possible to create a useful model within the first step. In all cases several iterations were necessary.
- Before a PIM to PSM transformation, the models must always be verified and validated. There's always a trade-off between the most flexible models for the business users and the highest possible accuracy of the models.
- When a model is changed, all runtime components must be rebuilt and distributed.
- Because of the MDD approach, the application is only as good as the specified model of the business users.

The article by Brown et al. [19] less is a classic case study as it is a collection of best practices, IBM has gained in the development of its own toolkit. The authors transform these into guidelines for the use of MDD and bring them into the case study.

4.4.4 Motorola

This case study by Baker et al. [9] describes the experience of Motorola with the model-driven software development. Motorola has already gained widely experience in several business areas with MDD. The models were created using UML 2.0, the subsequent transformations, however, conducted an in-house developed software, as none of the tools available on the market could meet the needs of Motorola. At Motorola 65%-85% code generation could be achieved. The development effort was decreased by a factor of 2.3 in the development, and by a factor of 30-70 in the correction of errors that were discovered during test. Seen about everything, the quality can be improved by a factor of 1.2 to 4 and the productivity by a factor of 2 to 8. Baker et al. observed the following points in their case study:

- System architects and designers tend to make implicit or explicit assumptions about the implementation of modelling.
- Many development teams were inflexible in changing the traditional development culture that was fostered by the absence of a defined MDD process.
- The third-party solutions scaled poorly and the generated code was inferior to the self-programmed solution.
- There is no development environment, which would cover all the needs of Motorola.

In conclusion, all of the listed case studies show that manual coding is still necessary. The statement that there is no viable all-in-one solution for the development of MDD projects closely follows this observation. Overall, apart from ABB [94], the use of MDD in software development was rated as positive and saved a lot of effort by the application of this paradigm.

4.5 Summary

The basic ideas and principles of model-driven software development as described in the MDA Guide (OMG) [68] are already widely used in practice [61]. An established and uniform MDA/MDD-process in which policies and processes are defined in a standard form, one looks in vain. The possibilities are so wide that a consistent process model can be developed only gradually. However, there are some more or less mandatory resulting activities, artefacts and roles that can also be found in the studied process models.

The benefits of the MDD-based approach can also be shown without an established process model and are also described in the various case studies: A higher level of abstraction in the domain-specific model increases the expressive power of the business models. There are also shorter development times and higher productivity, with lower project risk. In addition, media breaks are eliminated, which cause in the traditional development unnecessarily high costs. But productivity is not the only benefit, but also the higher quality results. Both the domain-specific language and the generator contribute to an improved quality. These statements are confirmed by a survey of the computer science research centre FZI in Karlsruhe [41]. 93% of respondents would use modelling and generation in future projects again. 80% of them see the high development rate as a benefit, 71% the clean source code, 68% the higher quality. However, only 37% regard MDD as a more cost-effective approach.

But the disadvantages are the higher complexity of the development of the DSL (metamodels) and the corresponding model transformations (generators) with the associated high costs. This effort, in conjunction with the given complexity, the lack of standardized processes and the prior unpredictable profitability makes decisions against

the model-driven development easier to understand. And in the survey of the FZI indicate 40% of the respondents who want to use no more MDD, that the approach takes more effort than benefit.

5 Approach: Using Agile Elements in MDD-Processes

5.1 Combination of MDD and Agile

An iterative approach in the development of the "Domain Architecture" and a stronger integration with application development is described in a case study in chapter 4 as a possible improvement. As a result, parts of the "Domain Architecture" would be available for application development earlier, and thus functional final results could be achieved earlier too. This could also promote the exchange of experience between application developers and MDD Infrastructure developers. The early provision of results and the close cooperation of all parties involved are essential characteristics of agile process models such as Scrum, XP, etc. The same applies to iterative development in short cycles. Therefore, it is only natural to apply these agile concepts to model-driven development.

However, before agile working techniques and principles can be used in model-driven development, it is necessary to investigate which project phases are important within model-driven development. In other words: Which substeps are important? Which artefacts are relevant? And which team roles are required? The starting point for this can be the process models for model-driven development identified and described in Chapter 2.3. MODA-TEL, MASTER, etc. are based on classical process models, but provide valuable information. The common features of the existing procedural models can be worked out and presented as a metamodel on a more abstract level. When defining a new and agile process, these process elements can be reused and recombined.

The basic idea is to make the new process agile and thus achieve the characteristics and goals described above. In addition, the question arises as to which agile working

techniques and principles should be applied. Similarly, the question arises as to which fundamental agile process is the basis for a new process model for agile model-driven development. On this basis, the necessary process steps, artefacts and team roles can be defined.

5.2 A detailed view on MDD methodologies

Starting point for further considerations are the identified development processes for model-driven development projects. Even if these are incomplete and imprecise, as mentioned in 2.3, they describe the necessary actions and relevant result types of model-driven development. For this reason, the similarities of the various development processes were identified and its elements described in a metamodel. The following Fig. 24 shows an excerpt of the metamodel and its derivation from a MDD process (here MODA-TEL, [42]). The metamodel thus describes the building blocks, which can be used for a new process or as a supplement to an existing process.

In this way, it is possible that elements of different process models become comparable. In addition, the gaps in the processes will become more apparent, and it is clearly represented what elements are used by a specific process and which are not. And ultimately, this general description allows a specific tailoring to a derived process.

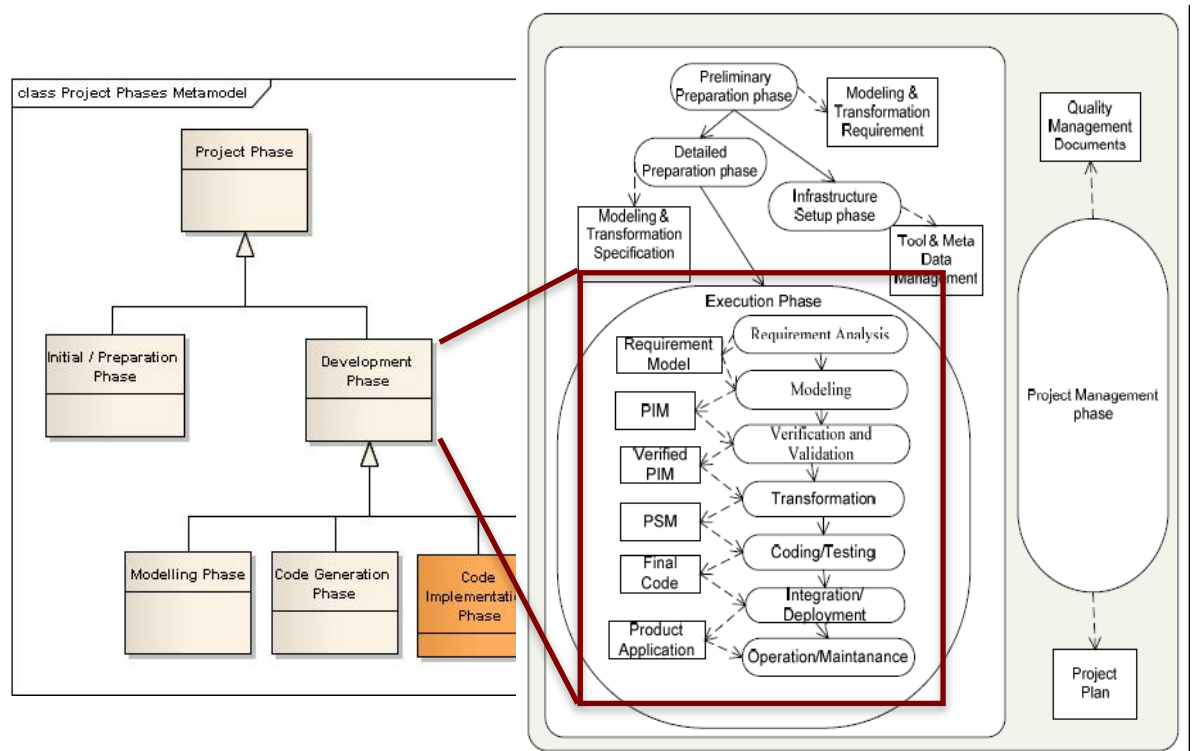


Fig. 22: Process Steps in MODA-TEL and the corresponding Element in the Metamodel

5.3 Commonalities of the Reviewed Methods

Based on the methods considered for MDD projects a basic metamodel was developed to describe the identified common process elements, result artefacts or roles of project participants and their relationships. The illustrated section of the metamodel on the next page (Fig. 23) shows the possible phases of the project as well as the artefacts, which are created by a team member in his role.

On closer examination of the mentioned methodologies, the following common elements of the methodologies can be derived: project phases, roles and development artefacts. In some methodologies, the project phases are only classified in “Analysis Phase”, “Design Phase” and “Implementation Phase”, and are reminiscent of the corresponding stages in

conventional development processes. In these cases only the results of the phases are related to model-driven development. So, for example, the analysis phase in the ODAC-method provides the platform independent model (PIM) as a result.

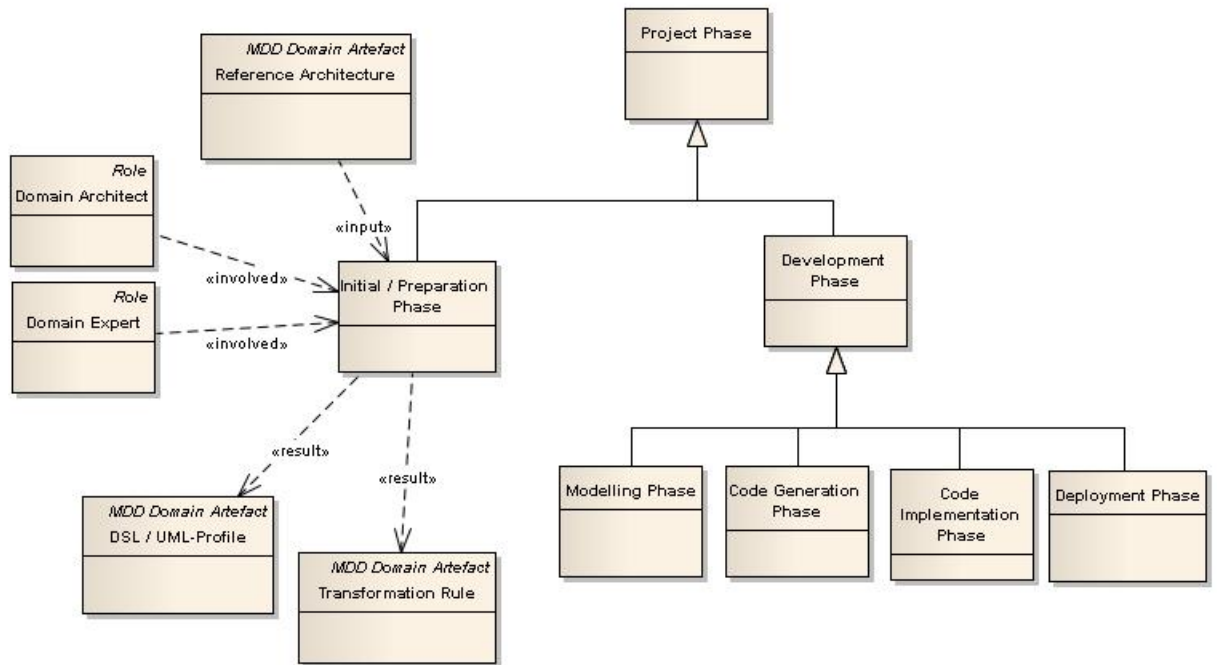


Fig. 23: Metamodel with Process Elements

Other methodologies such as C³ or MODA-TEL, name specific phases related to model-driven development. Both, C³ and MODA-TEL, define an initial project phase that describes the requirements for the modelling and the necessary transformations. And, in addition, they define a software development phase, in which the real application development is done step by step, from the model-design over the code generation and the application deployment. In addition to C³ and MODA-TEL, an initial project phase can also be identified in the other named MDD methodologies. Once it is called “Engineering Specification” [43], another time “Standardization Phase” [52] or “Preparation Phase” [42]. In general, this phase includes the definition of the domain specific language via a metamodel, the definition of the corresponding transformations and the necessary tooling.

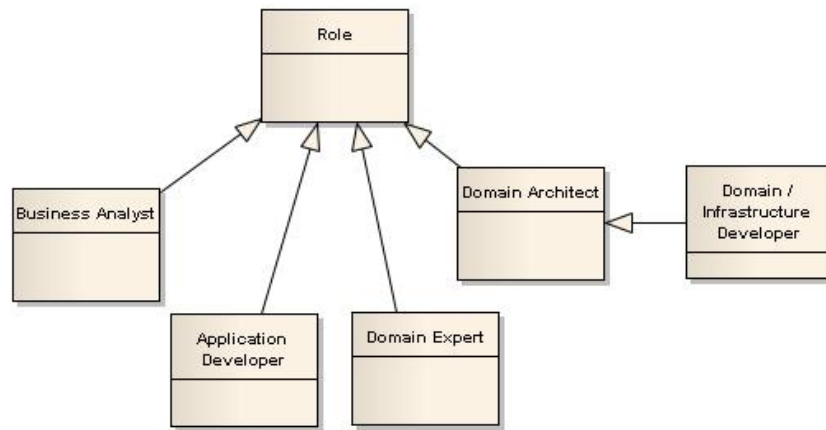


Fig. 24: Metamodel: Roles in Existing MDD Processes

All named methodologies describe specific roles for project participants very rarely. However, a description of the necessary skill requirements and responsibilities for those involved in the development process would be important. A good description of roles can be found in [91]. The authors separate the developers of the MDD-infrastructure (e.g. the “Domain Architect” or “domain expert”) by the users of the MDD-technology (the Application Developers).

5.3.1 Project Phases and Steps

The basic process within the framework of the model-driven development always consists of a phase of preparation (the initial phase). The authors Stahl & Völter call this in [91] the development of the Domain Architecture (explained in chapter 2.2.3). The domain-specific language and the associated metamodel are developed on the one hand. However, the necessary transformations for the transfer of the models to the source code are also derived using a reference implementation. This also creates the necessary toolset and the programming model for the development. This phase is influenced, on the one hand, by the professional environment with its terms and its structure, from which the

professionally motivated domain-specific language is derived. On the other hand, the chosen application architecture, which ultimately determines how the DSL language elements are transformed into technical artefacts. Thus, these are influences that are determined in classic projects within the framework of a specification phase and are described as functional and non-functional requirements. From the point of view of an agile software development the inclusion of these requirements and therefore also the development of the DSL and the architecture must be designed evolutionarily.

The second phase in a project with model-driven development is the real development of the application (development phase). The domain-specific language is used for the creation of models and the professional domain is described. Code generators translate the models into source code according to the specified transformations. Since the code generation is usually not complete, this development phase is typically additionally implemented manually. These two phases are explicitly found in MODA-TEL [42] as well as Stahl and Völter [91]. When projected onto an agile model, these two phases appear to be the most appropriate feature-driven development [24], where a rough overall model is refined and modeled in later iterations as well as modeled and encoded (see 0).

While in the described development processes these phases do not overlap, this will in the use of agile approaches necessarily be the case. Therefore, the definition of the Domain Architecture must be carried out in an iterative and incremental way, and in parallel to the application development. It should be noted that, as described in section 5.6, the architecture of the target system is previously defined and a tailoring of a reference architecture has occurred. Hence the agile principle of "system metaphor" is supported, which requires that all developers know and understand the basic architecture of the system.

The phase of the actual development follows the integration and test phase as well as the delivery for all mentioned process models. With agile aspects, this should be done as frequently as possible and at short intervals. However, this represents a special challenge to the development environment and the toolset.

The necessary process-accompanying quality assurance is described in little detail in all process models. Only in MODA-TEL is a model verification and validation proposed in the context of development (after modeling and before transformation).

5.3.2 Artefacts and Result Types

A complete description of a development process includes besides the explanation of the process steps the explanation of what types of results or artefacts are created and at which time. In this context, the defined term “Domain Architecture” [91] has been described in section 2.2.3. It describes the main artefacts that can be understood in its broadest sense as infrastructure components of the development environment for model-driven projects. Without these artefacts, including the definition of the domain specific language (DSL) or the transformation rules for models and generators, can be performed no model-driven project. Though Stahl et al. don't refer to a specific process model for model-driven development, also require process models like DREAM, MODA-TEL, ODAC etc. the development of these components, and define appropriate activities.

Identified artefacts of model-driven development are:

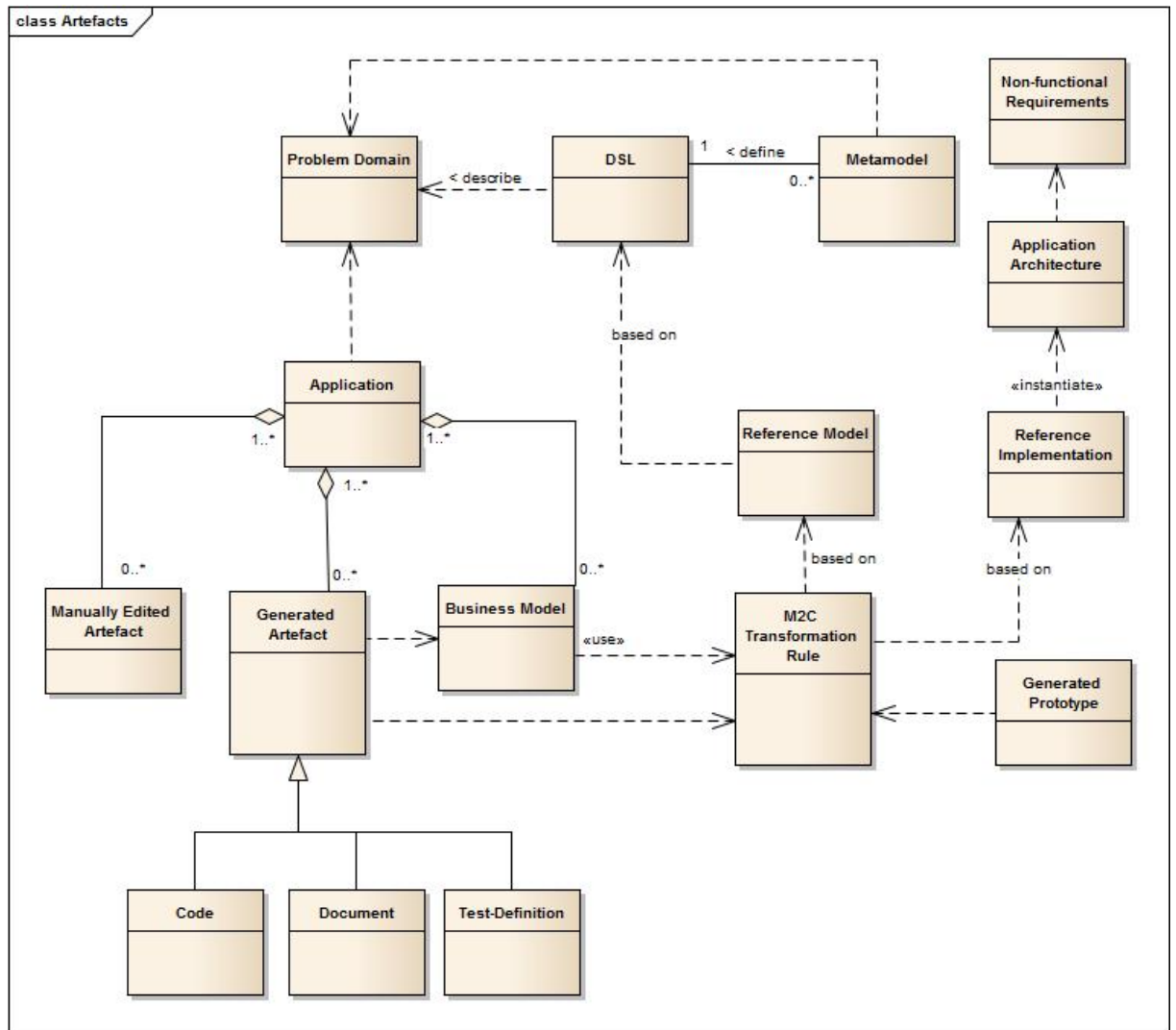


Fig. 25: Artefacts in the Model-Driven Development

- The **DSL (domain-specific language)** consists of individual language elements, which originate from the problem domain and are suitable for describing this problem domain. The DSL is described by a **metamodel**. Model elements in the metamodel define the language elements of the DSL. In addition to the basic membership of a UML element, each language element has further properties that allow additional information to be recorded. Relationships between the model elements in the metamodel represent how individual language elements of the DSL are interrelated and used.

- The **application architecture** defines the structure and relationships in the future application system. The non-functional requirements specify the framework conditions for the architecture. A **reference implementation** or a **minimal prototype** ensures that the defined architecture requirements work and thus enables risk minimization. If the architecture is created evolutionarily, it must be ensured that the reference implementation is further developed accordingly.
- Based on the reference implementation the necessary **technical infrastructure** can be built up. The technical infrastructure is, on the one hand, the environment for deploying and testing the application. At the same time, however, the necessary toolset for the development can also be defined. The basis for this is the reference implementation based on the defined application architecture.
- Using the domain-specific language a **reference model** is created, which demonstrates the use of the language elements of the DSL. This reference model is the basis for the derivation of the **transformation rules** by means of which the reference model can be transferred to the reference implementation.
- In most cases, the transformation to source code will only cover part of the reference implementation and will require **manual implementation** of the missing parts. Inserting the missing functionality requires additional architecture requirements for the development. Appropriate patterns (for example, Strategy or Factory) can be used to ensure that the manual parts are coupled as loosely as possible to the generated elements.

5.3.3 Roles and Team Members

A description of the roles of involved project members is also part of a complete description of a process model. But in this regard, the descriptions of the identified processes like MODA-TEL etc. are incomplete. Only in Stahl et al. [91] is a description of roles included, which differ roughly between Domain Developer and Application Developer.

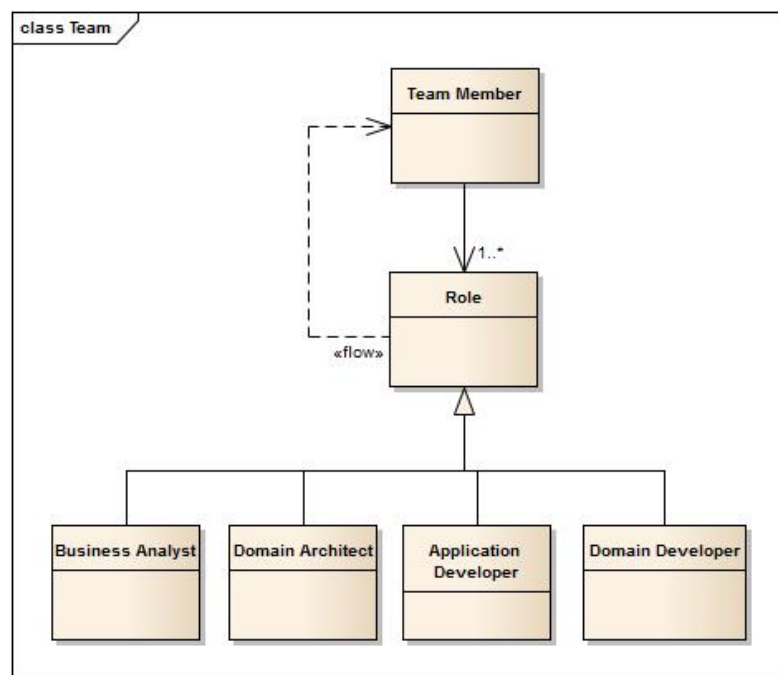


Fig. 26: Identified Project Member Roles in MDD-Projects

From the point of agile methods additional role definitions must be considered. For instance, roles from the FDD process such as chief architect, feature team or chief programmer are conceivable and must be assigned to the individual process steps. It must be considered however, that the agile principle of a self-organizing team doesn't become limited or overloaded by too many roles.

The following roles can be identified in the context of model-driven development:

- The **Application Architect** defines the application architecture according to the non-functional requirements and directs the development of the reference implementation, which follows the requirements of the application architecture.
- **Application Developers** implement the reference implementation and later, within the framework of the application development, the code portions to be generated manually.
- The **Domain Architect** defines the domain-specific language using a metamodel.
- A **Business Analyst** describes the elements of the domain and thus supports the definition of DSL by the Domain Architect. This person creates a reference model with the help of the DSL and thus describes a section from the professional domain. In the actual application development, the Business Analyst will model the technical requirements and coordinate with the Application Developer about the parts to be implemented manually.
- **Domain Developers**, together with the Domain Architect and Application Architect, define the transformation rules for mapping to the reference implementation. The missing and manually coded portions are implemented by the Application Developer in coordination with the Domain Developer and the Application Architect.

Agile practices that have established themselves in the context of modelling and seem to be suitable for a process model for MDD are now considered below.

5.4 Agile Techniques and Practices for Modelling

Based on the large number of agile methods and techniques such as XP, Scrum, etc. it is to identify individual procedures that are suitable for creating the artefacts described in section 5.3.2. For this, it must be a fundamental distinction between agile techniques such as “pair programming” and “continuous integration” and agile process models such as Scrum, FDD, etc. (see Section 3.1.3).

In particular, the application of agile techniques, however, has an impact on the model-driven process. This becomes particularly clear in the application of refactoring, which is leading to frequent changes on the created artefacts. This may play a minor role in the editing of the artefacts of the application development, however a refactoring of Domain Architecture components has a significantly larger impact.

So here is clearly to define which phases of the development interdigitate, and which artefacts can be developed incrementally, which dependencies are acceptable and for which areas clear guidelines are required.

In the selection of appropriate methods for supporting model-driven development will therefore initially have worked out those agile techniques that are suitable for the preparation of the respective artefacts. In the field of modelling have been already identified some agile modelling techniques in the elaborations of Ambler [5], Cáceres et al. [20], Baker et al. [9] and Pei-Breivold et al. [77], which can serve as a starting point. The following table represents a selection of agile modelling principles:

Discipline	Agile practise / technique
------------	----------------------------

Modelling	Assume simplicity / simple design
	Architecture envisioning
	Model storming
	Just barely good enough
	Iteration modelling
	Multiple models
	Document continuously

Fig. 27: Agile Modelling Principles¹

In the following the different agile techniques will be analysed in terms of their suitability for elements of the model-driven development. For this, some criteria have to be defined, by which agile techniques can be evaluated. These criteria are:

- What artefacts are affected?
- What is the impact of the agile technique on the development effort and how to the temporal aspect?
- In which phases of the project, the technique is applicable?
- Do dependencies to other artefacts exist? What artefacts are indirectly affected?

5.4.1 Assume Simplicity / Simple Design

This principle first of all advises that the simplest solution is the best solution. With regard to agile modeling, this means that only these properties are to be modeled, which is necessary for the current state of knowledge and for the current task. It is advised to concentrate on the existing requirements and to rework the model successively through

¹ <http://www.agilemodeling.com/principles.htm> (checked on 10/09/2015)

refactoring.

With regard to the model-driven development, this can, of course, primarily be projected onto the business model, which is described with the DSL. Iteratively, this model can be extended and refactored. But does this also work for the description or definition of the DSL, the metamodel? The experience from the case studies (see chapter 4) suggests that complex and extensive DSLs lead to problems with dependencies and increased risks for the project.

Artefacts: Metamodel, Business Model

Project phase: Initial Phase, Development Phase

Dependencies: -

5.4.2 Architecture Envisioning

Architecture Envisioning is an agile practice of developing the application architecture at a high abstract level and discussing the technical implementation with the team on this basis. The goal is to develop a strategy for the architecture [4] instead of creating extensive documentation. The architecture is then further developed during application development in model storming sessions. This can be transferred to the model-driven software development. If a fundamental strategy of an application architecture has been developed, the corresponding reference implementation as well as the necessary model-to-code transformations can be derived on this basis. The revision of the architecture then leads to corresponding changes to these artefacts. Especially in this aspect is a great strength of the model-driven development, since the architectural changes can be implemented by the push of a button.

Artefacts: Application Architecture

Project phase: Initial Phase, Development Phase

Dependencies: Transformation Rules

5.4.3 Model Storming

Scott Ambler called Model Storming in [4] as just in time modeling. The basic idea behind this is to solve problems by spontaneously forming a team of team members who can help. The author points to work techniques from Extreme Programming (XP), such as Stand-up design sessions [54]. However, the emphasis is on drawing models as sketches on paper or whiteboards. The work technology therefore derives its advantage from the simplicity and spontaneous changeability during the discussion. In this way not only models, but also screen sketches or handwritten CRC cards are created. Thus, the results of this working technique are not to be processed in machine form. Subsequent transfer of the sketches into actual models seems to contradict the agile approach. For this reason, from the point of view of model-driven development, this technology is either not applicable or difficult to apply.

5.4.4 Just Barely Good Enough

Behind the "Just barely good enough" principle or JBGE is the statement that one should avoid unnecessarily much effort to invest a partial result or artefact, which in a simple form already fully fulfills its purpose. This does not mean that quality is lost. The focus is on delivering precisely the required quality and scope as required - but not more. Agile modeling is often sketched by hand, as this is often sufficient for communication and discussion of the facts. From the point of view of model-driven software development,

however, the principle is also applicable and can be specifically related to the extent and scope of the DSL. This should be able to describe the necessary facts of a problem solution, but it should not be possible. In practice it is often observed (see chapter 4) that domain-specific languages are frequently overloaded with additional features (such as additional tagged values), which are not related to the problem itself. They usually provide technical information for a simpler model-to-code transformation, but they violate a fundamental principle: separation of concerns [32] or the single-responsible principle [66]).

Artefacts: Metamodel, DSL

Project Phase: Initial Phase

Dependencies: Transformation Rules, Business Model

5.4.5 Iteration Modelling

Iteration modeling goes hand in hand with the principle of Just Barely Good Enough (see 5.4.4). At the beginning of each iteration, a model sketch should be created, for example the definition of a data structure or the sketch of a screen. The model helps to describe the complexity and scope of the iteration. As a time frame for an iteration, a typical time span of two weeks is given for agile methods. The model must be sufficiently precise, so that the effort can be estimated and the work planned (JBGE). Iteration modeling is therefore a technique to support the effort and therefore interesting for the project planning, but not for the concrete creation of the artefacts in the model-driven development.

5.4.6 Multiple Models

In the case of multiple models, the basic assumption is that the knowledge of various modeling techniques and diagram forms is necessary for effective and meaningful modeling. The modeling by means of different diagram forms enables the representation of a situation from different perspectives (Views) and at different levels of abstraction. As a result, different stakeholders can be addressed and issues can be made more comprehensible and understandable. However, using different modeling techniques is not easy. There are simply too many different diagram types. The UML alone has 13 diagram types, and Scott Ambler lists 42 different diagram forms on agilemodeling.com².

There are two aspects to the model-driven development: For the Domain Architect, who designs the DSL, it is important to know which diagrams best describe the problem domain. The Domain Architect must decide on which basis the DSL is developed. For this, a deep knowledge of the individual modeling languages is necessary. For the user of the DSL, the Business Analyst, it is really only important that this person knows the language means used in the DSL and can use them correctly.

Multiple models makes it possible to describe different views with the correct language means. This is not only useful in the model-driven development, but in general.

Artefacts: Metamodel, DSL

Project Phase: Initial Phase, Development Phase

Dependencies: Business Model

² <http://agilemodeling.com/artefacts/> (checked on 01/12/2016)

5.4.7 Document Continuously

The goal of any agile method is to have a potentially deliverable or even applicable product at the end of each iteration. Of course, this documentation also includes a documentation of this product. In traditional process models, a large part of the documentation is produced at the beginning of the project in the form of a project plan, a detailed requirement analysis or a design. After that, the documentation is stopped and is supplemented by support documents or user manuals only towards the end of the project.

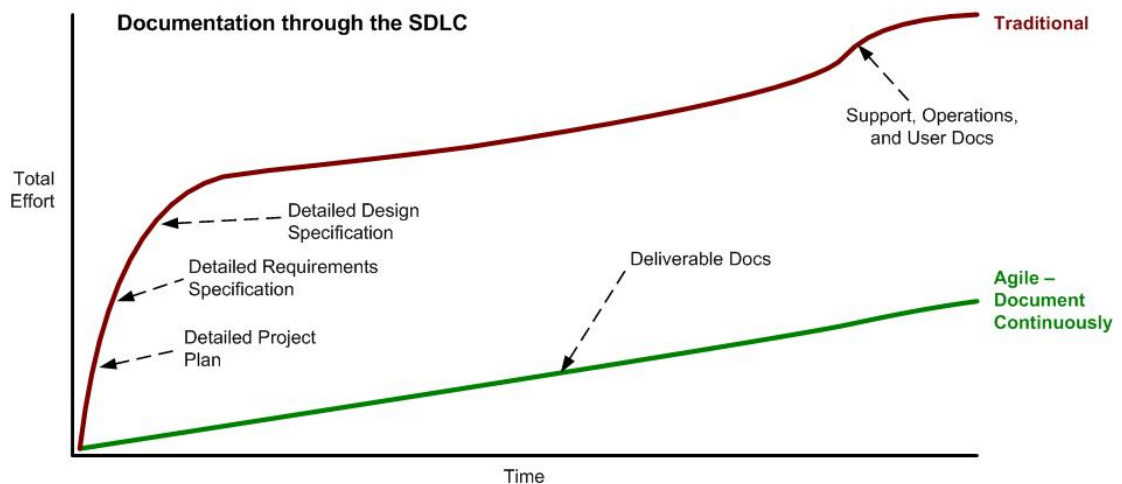


Fig. 28: Document Continuously ³

In agile process models, the documentation is not always in the foreground. In the agile Manifesto [2], working software is preferred to extensive documentation. Therefore, the scope of the necessary documentation is discussed in [4] and [82]. This includes all documents relevant to the stakeholders as the addressees of the delivery. These are therefore less specification documents (or models), but rather user manuals, deployment, and system documentation. These documents are best prepared in the following iteration (for short iteration cycles) or within the iteration (for long iteration cycles) [82].

³ <http://www.agilemodeling.com/principles.htm> (checked on 10/09/2015)

For the model-driven development, the question arises as to which documentation can be created and delivered based on the models. The application architecture responds to this. It depends on which artefacts are relevant for the delivery of a functioning and installable software. In addition, all models that describe interaction scenarios with the application system to be created are of interest to users, and thus provide the stakeholders with an idea of the scenarios that have already been implemented. This can be useful in the context of early quality assurance.

Artefacts: Application Architecture, Business Model

Project Phase: Development Phase

Dependencies: -

5.4.8 Some other Practices

In addition to the above-mentioned practices from agile modeling, there are other working techniques from the agile context that can be used for an application in the context of an agile model-driven development. These include:

- **User Stories:** User Stories [27] are software requirements formulated in everyday language. They should be deliberately kept tight and formulate their statement with a maximum of two sentences. Similar to use cases, user stories represent the requirement in the language of the user and thus offer a good starting point for the modeling. In agile modeling, a user story could face a corresponding, scarce model.

- **User Story Mapping:** The Story Map [76] is a workflow that allows users to graphically represent the user's successive activities in a graphical overview. In addition to this, the customer stories, epics etc. up to the individual user stories are displayed vertically. This representation can also contain a corresponding simple model.
- **Reviews and Retrospectives:** In many agile action models (such as Scrum), reviews and retrospectives are performed at the end of an iteration (in Scrum Sprint). Reviews are used to present the results of an iteration and thus the quality assurance of the content. Retrospectives consider the procedure, the observance of the process and the teamwork and are to contribute to the process improvement. Both techniques should also be part of the agile model-driven development.

This list of agile working techniques can be continued for a long time, and in addition to the theoretical approaches, some additional working techniques have developed. Usually as a result of an adaptation of existing approaches to an individual development process.

5.5 The Appropriate Agile Development Process

Comparable to the evaluation of agile techniques in terms of their suitability for model-driven development processes must also be a review of the existing agile process models. Finally, if a development process should be defined to support model-driven development (e.g. the ODAC process) with agile methods, the following must be certain: On one hand must be known, which agile processes in their orientation are close to the model-driven development processes. And secondly, to what extent they are suitable with their process flow and how they cover the requirements.

A starting point for the determination of relevant agile processes is the selection in section 3.3, which is based on a study [78]. Here agile processes have been identified that meet the needs of model-driven projects most likely because they have their priorities in modelling, requirements management and system design. Based on this selection are sufficiently information available, to define a suitable process for MDD, or to identify single process elements for the integration in an existing MDD process (i.e. the ODAC process).

A further starting point for the selection of an agile method as the basis for the definition of an agile model-driven process can be the study "Status Quo Agile" [62] of the University of Applied Sciences Koblenz, in which more than 1,000 international participants were surveyed on the use, spread and success of agile methods used. The results can be summarized as follows:

- Mixed form or pure form: The majority of users of agile methods use these only selectively or in a mixed form. The consistent use of agile methods is only the case for approx. 25% of users.

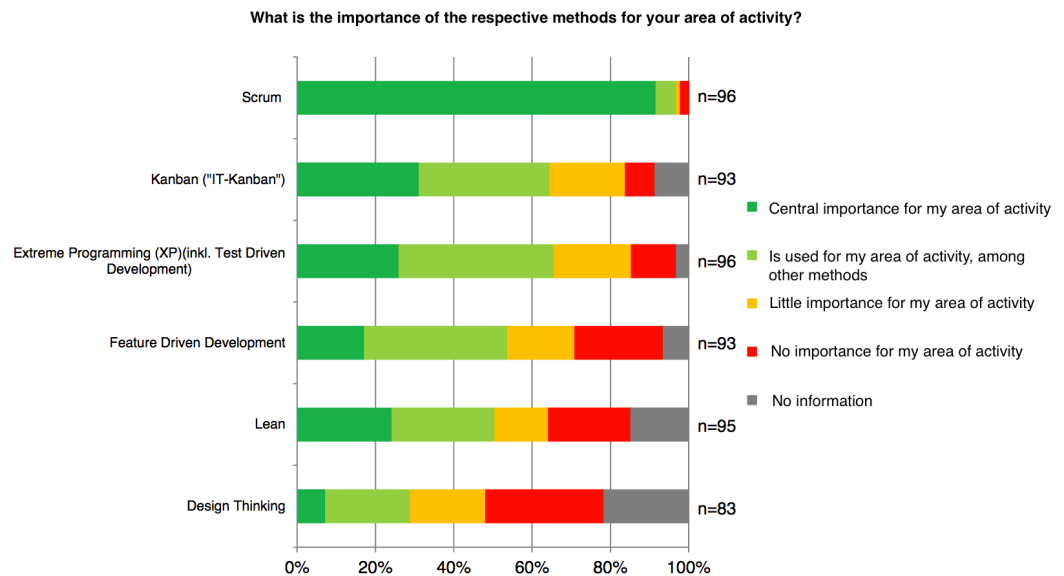


Fig. 29: Important Agile Methods [62]

- Scrum is the leading agile method: Scrum is the most widely used agile method and is applied by 86% of respondents. Then follow Kanban, XP and Feature Driven Development. This can also be projected on the applied agile techniques. 85% of the seven most common techniques come from the scrum environment and nearly 70% of the 22 specifically requested techniques were used by at least 70% of the users. And also in the assessment of the successes, Scrum is rated better in each sub criterion than other methods.

The study also contains figures on the size of the project teams or the typical iteration duration.

- The typical team size is given by 5-9 people. Interestingly, this is also the most common team size for classical users.
- The duration of a sprint is specified by 2/3 of the respondents with a maximum of three weeks.

Overall, the success rate of agile methods is assessed more positively than the classical methods. Thus, agile methods are also better evaluated in partial criteria such as "quality of results", "employee motivation", "efficiency" or "adherence to schedules". Only 6% of users of agile methods and 10% of users of classical methods call agile project teams as undisciplined.

For the definition of an agile model-driven process, it is therefore useful to reuse basic concepts and structures from Scrum and thus to place the broad spread and the high degree of recognition. Techniques from XP and Feature Driven Development can certainly be considered, since these are also mentioned among the most frequently used agile methods.

5.6 The Meaning of Architecture in Agile Projects

In model-driven projects, defining the architecture of the target system is an important step in the development of the Domain Architecture. The architecture of the target system is the basis for the derivation of the transformation rules. This determines how a model element is transferred to a target architecture-compliant implementation in source code. Due to this great relevance of the architecture for the model-driven development will be

often spoken of an architecture-centric approach [91]. Therefore, in the context of this research is to clarify, which significance the term "architecture" has in an agile environment. And also, what best practices have proven successful for the development of architecture.

In their daily work IT professionals often use the term “architecture”. As part of the development of software solutions they paint graphics for different levels or views with boxes and arrows, and call the result “architecture”. In daily practice, the message of the graphics is often intuitively clear and describes closely the necessary information for the development. From the perspective of quality in a project, you are mostly on the safe side. But if it comes to concrete quantitative efficiency of the approach in the development, there is the question of how much architecture has to be meaningfully defined. How much architectural specifications should be given to the development team, so they do not build something, which must be fundamental rebuilt at the end with high effort? Which part of the architecture can be omitted, since it is so trivial either that it will be built in any case like this, or because it limits the creativity and ultimately the effectiveness of the developers? How important is the definition of an architecture in the agile development process, and how much can be left to the self-organizing team?

5.6.1 Architecture - a Definition

In this context, it is important to have a clear definition of architecture. But due to the lack of clear terms in this discussion, terms such as "architectural style" or "reference architecture" will appear. In contrast, the definition of the term "architecture" should be defined much more sharply. For this should be considered once, how to use the term "architecture" in the construction industry. In this context architecture describes the use

of certain types of components as well as a certain way to use these components. Transferring these considerations to the domain of the software systems the following definition of architecture can be deduced:

Architecture describes the basic organization of a system by the kinds of its components and the kinds of relationships between these components, or the way in which they interact together. The definition of the architecture includes hence the design principles of the system.

What do the terms component and relationship mean in this context?

- The term "component" refers to part of a system in terms of general systems theory. A system consists of parts - and these are called components. The whole system is described, if, firstly, its components are described with their properties and, secondly, the relationships in which they are related.
- "Relationships" between components can be interpreted more generally. It's not just about the static relationships but also about dynamically changing relationships and the dynamics of the overall system as well as the interaction between components.

By this definition, architecture describes getting a whole class of systems and never a single system. Moreover, by this definition, the architecture of a system is clearly delineated from the design of a system. In architecture only the types of components are of interest. In contrast, in design each component and possibly also their internal structure and specific behaviour are of interest. This is referred often as detailed design or

component design. To verify this interpretation, the following IT architectures are classified and defined:

- SOA: The acronym stands for service-oriented architecture. The popular definitions do not argue about types of components and relationships and are correspondingly uncertain. In [37] the SOA-term is defined as follows: *“The types of components are those which can be identified on the basis of the services of the business and business strategy, and the way of interaction is the “appropriate loose coupling”.*
- Web service architecture: The types of components are service requestor, service provider and service registry. The way of the interaction is based on the pattern of "publish, find and bind".
- Model-View-Controller: The component types are model, view and controller. The kind of interaction is: controller to model and model to view, with the latter one usually follows the observer pattern.
- Web application: The types of components are client components on the basis of web browsers and server components based on web- or application servers. The interaction happens directly or indirectly via HTTP.

These examples show that the definition of architecture in the domain of software systems seems to fit very well.

5.6.2 The Difference between Architecture and Design

An important issue for the benefit of this architecture definition is due to the clear distinction between architecture and design. A well-made design of a software system results in particular from business functionality – these are the functional requirements. An adequate breakdown of the functionality can be found in the ideal case 1:1 in the structure of the system back to its individual components. And also the internal structure and the specific behaviour of the components are determined by the particular functional requirements. However, the architecture of a software system is derived from the non-functional requirements. The defined types of components primarily result from the requirement for long-term maintainability, extensibility and modifiability of the software.

This relationship to non-functional requirements can also be shown using the examples above. In an SOA, for example, the non-functional requirement is the alignment of IT with the business and the opportunity for flexible adaptation to changing business processes. And a web application follows the non-functional requirement of replacement of software distribution.

Software architecture is therefore ideal when the amount of all prioritized or weighted non-functional requirements are optimally met. Ultimately, this is always a plan-specific and adequate trade-off. A very similar approach is also the base of methods such as ATAM (Architecture Trade-off Analysis Method) of the Software Engineering Institute [87]. The architecture development should consider all potentially conflicting non-functional requirements explicitly, and considering the trade-offs strive for the optimum. These trade-offs are different in each concrete development project, and for the necessary decisions an architect needs a lot of experience. These considerations lead now with regard to the development efficiency on to three key statements and best practices:

- Define architecture in advance
- Architect should be a team member
- Tailor the reference architectures

Below these three best practices are considered in more detail.

5.6.3 Agile Best Practices for Architecture

5.6.3.1 Define Architecture in Advance

An iterative approach often proves to be the best way to precisely record requirements. It does not attempt to understand in advance all the requirements and to create a complete specification or a detailed design of the system. Instead, only parts are recorded accurately and then implemented directly into software. The existing software helps the stakeholders to describe the requirements for the system in the next iteration better. In sum, this approach is often efficient.

However, it is different for non-functional requirements. The experience shows that it is most efficient to get this understanding as well as possible in advance. Learning about the non-functional requirements in iterations is only in exceptional cases a good approach. And then the iterations usually have the character of prototypes for architectural evaluation. Mostly, it is much more efficient, to analyse the non-functional requirements in advance and to identify the trade-off and to derive the architecture from it. This is especially true because the architecture defines the types of components and any change or extension of the architecture may lead to extensive renovation of all previously developed specific components. From experience, this is rarely effective. In this sense, this understanding of architecture also covered by the definition of Grady Booch: “*All*

architecture is design but not all design is architecture. Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change.” [16]

Also for agile - and just for efficient projects is therefore the logical claim: To develop the architecture of the software system in advance as far as possible and not in the context of iterations.

But this sounds like a contradiction to the agile approach. Finally, the agile manifesto [2] emphasizes, "*The best architectures, requirements and designs are of self-organizing teams*". In established agile process models like Scrum [86], this means a development within the iterations. With the sharper definition of architecture from above, this is not longer true. Even if still some improvements to the architecture are possible, the essence should be pre-thought, especially in the light of the above considerations about efficiency and to avoid refactoring [39] in the agile process. The design, however, may and should develop iteratively - emergent design, as part of agile software development, is not a contradiction to the advancement of the architecture definition. This advancement of the architecture definition can be used with popular iterative approaches - be it through a preceding basic concept for the system in a controlled, incremental approach or a sufficiently extensive envisioning phase in the Scrum project. In no case there is an anti-pattern.

5.6.3.2 Architect Should Be a Team Member

It is important to separate the basic procedure in the development of architecture, design and implementation intellectually from the distribution of tasks and roles in the project

team. In Scrum, for example, the development in sprints takes place, carried out by the self-organizing team.

It has already been established that the architecture definition should be performed before the iterations. In addition, the question arises to what extent the support of an experienced architect for the efficient implementation of the architecture is required. In Scrum, for example, there is no such role. In Scrum, the self-organizing team is responsible for all content-related tasks, in keeping with the previously cited quote from the agile manifesto. The above explanations, however, show clearly that the architecture definition is an essential phase, defining the success of the project, and that due to the inherent complexity of the trade-offs an extensive experience is required.

Hence for agile and especially for efficient projects it is imperative: Make sure you have a designated architect in your team.

Whether this architect now gets the explicit role as "chief architect" and thus explicitly the corresponding responsibility or whether the architect act as "primus inter pares" in the self-organizing team and does the job because of his experience, depends on further project characteristics.

5.6.3.3 Tailor the Reference Architectures

Reference architectures are given architectures, where you can be guided by the architecture definition in a concrete case. It is ideal in terms of efficiency when the target architecture can be composed largely of reference architectures, while each reference architecture can be reused easily without any other changes. Experience has shown that

this is rarely possible. Reason here is again the call for the architecture in terms of a reasonable trade-off.

An example: A high degree of decoupling and abstraction is needed especially when long-term maintainability, extensibility and modifiability have clearly the highest priority. This prioritization is often not questioned, but the given reference architecture is used unreflective. At the end, the development effort is high, and the developers are talking about over-engineering, because there were obviously competing demands. Without reference architectures software development is not efficient, but without the appropriate suitable cut also.

Therefore, for agile and especially for efficient projects applies: Work with reference architectures, but make sure they are in accordance with the trade-off considerations appropriately tailored.

For this, the following applies: Not using a reference architecture is still better than using a reference architecture that does not fit to the non-functional requirements. This also applies in principle to widespread reference architectures that have meaning in an agile environment. In fact, constellations are thinkable, in which a system explicitly should not be built according to a reference architecture. But above all, many constellations are possible, in which the reference architecture should be tailored to a lightweight architecture. For example, this can be done by simplifying the user interface without complicated dialogue structures, or, by not using interfaces or transport objects for decoupling, unless the implementation does not need to be replaced. When tailoring the reference architecture to the concrete architecture, the selected technology often plays an essential role. Many abstraction mechanisms are not often needed when a homogeneous

and modern technology, such as JEE5, can be used. A good example is the very simple software architecture in [15]. When using of JEE5 and under the simplifying assumption that the system solution may be mixed with the technology, a particular architecture can be selected, which differs in the complexity of the reference architecture by factors. It is obvious that less complex architectures for the team that is working on this basis can be particularly effective. If the tailoring of the reference architecture is adequate, the reference architectures significantly contribute to increasing the efficiency of software development.

5.7 Known Limitations

If agile techniques should be used in MDD projects, this has the consequence that also must be thought about the impact of agile techniques on the Domain Architecture. On the one hand it is possible to establish agile techniques in the phase of application development. As a reaction on changed functional requirements, platform independent models (PIMs) can be refactored like source code in conventional agile projects. But if there are new findings that affect the Domain Architecture (e.g. the metamodel), this will lead to accordant adjustments of the affected part of the Domain Architecture and all dependent artefacts.

Changes on non-functional requirements also have a strong impact on the Domain Architecture. But especially in agile process models there is the interest to identify and implement the requirements in several short iterations (e.g. Sprints in Scrum). Changes are welcome and the project reacts to this by refactoring the already developed artefacts. On the other hand, the early provision of results is a core postulation of agile processes. If a project team would like to deliver results as soon as possible, they can't wait until the development of the Domain Architecture is completed.

Therefore, it is an essential requirement to dovetail the two phases "development of the Domain Architecture" and "application development" in a way that they can be developed iteratively and incrementally, and the early delivery of parts of the application is made possible. Therefore must be determined that certain artefacts are created early on. This primarily includes the layout of the application architecture (see section 5.6).

5.8 Summary

As has been shown, the development processes in the model-driven development are essentially divided into two phases: a preparatory or initial phase as well as the subsequent real development. In the individual process steps within these phases, numerous roles are involved, which in turn influence the development of the DSL, but will also be involved in the application development on the other hand. Also, numerous artefacts are to be made, such as e.g. the reference implementation. This does not seem necessary at first sight, but ultimately it will ensure the quality of the model transformation into source code [17][91].

To achieve the above objective of an efficient model-driven development, the boundary conditions must be set for the definition of a development process, and also what kind of software development project should be considered.

Sommerville [89] judges about model-driven development that the use makes sense only if there are large, long-living systems, where requirement changes for the target platform is very likely during their lifetime. This is typical for business applications in the banking and insurance sector, although the desire for faster introduction of new software and a significant shortening of the software life cycle can be seen too. For this reason, even in

the selection of case studies, projects in the field of insurance applications were chosen. For further consideration within this research project, therefore, the focus is on the model-driven development of business applications in this area, which also possess the following characteristics:

- The considered projects are development projects of typical business applications. They are individually developed for the customer and the time frame is between 8 and 24 month. The team size ranges between 5 and 10 employees. So the necessary effort will be between 40 and 240 man-month.
- The application is typically distributed, component-based, multi-layered and based on a standard framework like JEE or a comparable technology.
- The application is developed new and isn't an extension of an existing application.

Having already identified possible components of a process model, there is now also a focus on a certain type of application system, on whose development a possible process should be oriented.

In addition, in this chapter, agile methods and techniques were first examined for their suitability for an agile model-driven process. The focus was on different agile modeling techniques. Some of these techniques can be interpreted in the sense of an agile MDD development or transferred to a corresponding procedure. This includes, for example, "Assume Simplicity / Simple Design", which must be considered both in the development of the Domain Architecture (ie in the initial phase in the metamodel) and in the later development phase. Especially in the complex environment of a model-driven development, the simplicity and manageability of a DSL is of great importance. And even

simple business models, which model concrete professional statements, can be interpreted as "user stories".

Another fundamental question is whether and on which agile method a possible process for the agile model-driven development should be based. Establishing an existing method is common practice. According to the study by the University of Applied Sciences Konstanz, agile methods are frequently adapted, and individual agile practices are also often used in mixed use. With Scrum as a possible basis, a candidate is given, which according to the study is the most widespread in practice and the most widely accepted. This also offers the opportunity for a new process to enable the developers to familiarize themselves quickly with familiar concepts. If concepts from Kanban or Feature Driven Development are required, this is also possible due to the high level of recognition.

The architecture of a future application is one of the foundations, especially in the model-driven development. From this, the mapping of the domain-specific language to the generated source code depends on the defined transformations. However, the importance and positioning of architecture in agile projects is often different. For this reason, a definition of architecture was made under the perspective of agile projects and individual agile best practices for architectural development were presented.

Thus, besides the properties of model-driven projects, agile methods and practices have also been considered for modeling. This is now the basis for the development of the Agile Model-Driven Method (AMDM), which is described below.

6 The Agile Model-Driven Method: AMDM

As learned in the studies of Asadi and Ramsin [7] and Chitforoush et al. [21], the existing methodologies for MDD projects are incomplete and their description is imprecise. Essentially they are based on traditional development processes, and also the process framework by Chitforoush or the development lifecycle of Asadi and Ramsin do not regard agile aspects. Other approaches like [5] focus on the use of models in agile methods, but they do not consider MDD.

Therefore, it is now the goal to develop an adapted process model for model-driven development projects from the results of the previous steps, which takes up elements of agile process models and uses appropriate agile techniques, i. e. adapted to the context of model-driven development. The process model is designed to support the model-driven development of business applications as described in chapter 1.2. As already mentioned in chapter 5, the existing MDD processes and their structure are a possible starting point for a new agile MDD process model, the Agile Model-Driven Method (AMDM). The identified elements of the process models and the selection of appropriate agile techniques for generating artefacts from model-driven projects provide the basis for optimization and adaptation.

6.1 Requirements and Constraints

The Agile Model-Driven Method (AMDM) organizes the development process into different structured phases and assigns suitable methods and agile working techniques to them. AMDM presents the tasks and activities required in the development process in a logical order. For this reason, AMDM is not only a process but can also be described as a methodology.

The approach is focused on the individual development of business applications that take between 8 months and 2 years to develop. The architecture of these applications is typically distributed, component-based, multi-layered, and based on a standard framework such as JEE or similar technology. Other frameworks (e. g. for persistence or UI) are also used. This type of application is comparable to the applications from the case studies in Chapter 4.

The aim is to develop business applications using model-driven development and to generate the source code automatically, thereby achieving higher code quality and increased efficiency. In addition, agile working techniques should be used and the process should be based on well known, established agile process models.

In this case, the typical steps of model-driven development, starting with the development of the Domain Architecture and the subsequent application development, will not take place successively but iteratively and incrementally. The aim is to deliver valuable intermediate results at short intervals.

6.2 Findings from the Studies

For the development of a new process model for agile model-driven development, the findings from previous research in this thesis will be used. This relates to the basics of model-driven software development from chapter 2. On the one hand, there are now findings about essential artefacts that have to be created in this context as an important infrastructure for development. These are primarily summarized under the term "Domain Architecture" (cf. Chapter 2.2.3). In addition to the tooling platform, these include important elements such as a metamodel for describing the domain-specific language (DSL), the transformation rules for model-to-code transformation, etc.

In addition, existing process models for model-driven development provide insights on the essential phases and activities of model-driven development. The methods ODAC (cf. Chapter 2.3.1), MASTER (cf. Chapter 2.3.2) and DREAM (cf. Chapter 2.3.3), which are mainly used in the scientific environment, are hardly applied in practice and cannot be classified as agile process models. Nevertheless, they provide important information for AMDM via common phases, work steps and generated artefacts.

In addition, the criticism of model-driven development, e.g. by Heijstek and Chaudron [49], shows that, for example, model-to-code transformation is to be regarded as an additional application in development. By providing this application and the necessary infrastructure, additional activities, a higher complexity and a high initial effort necessarily arise (cf. Chapter 2.4.1 and Fig. 5). This is not reflected in the investigated process models. The authors Asadi and Ramsin [7] also share this opinion in their review of model-driven development processes. AMDM integrates these working steps with regard to the infrastructure and reduces complexity and initial effort through an iterative approach. These challenges and points of criticism can also be verified by the case studies,

which originate from projects in the author's working environment as well as from other case studies in industry (cf. Chapter 4). These include, for example, the high initial effort, complexity and effects of changes.

Chapter 3 has provided insights into what working techniques exist for modeling from an agile point of view. From existing approaches such as Feature-Driven Development, concepts are integrated into AMDM. AMDD by Scott Ambler describes short modelling cycles and working techniques such as Model Storming, Iterative Modelling or Initial Architecture Modelling (see chapter 3.4.1). The MIDAS framework is based on continuous integration and different viewpoints (see Chapter 3.4.2). In addition, it was worked out that AMDM must create the possibility to react to changes in both technical requirements and business requirements (see Chapter 3.5, Fig. 14). AMDM takes this into account by introducing different sprint types and parallel development branches.

In order to create a basis for the definition of AMDM, Chapter 5 first of all worked out the commonalities of the examined process models for model-driven development and described them using metamodels. These include the different project phases (cf. Chapter 5.3.1), artefacts to be created (cf. Chapter 5.3.2) and the roles of the necessary team members (cf. Chapter 5.3.3). In addition, agile working techniques for modelling were considered in Chapter 5.4. It was shown for which phases these working techniques are suitable, which artefacts are affected and which dependencies exist. Finally, chapter 5.5 stated that the agile procedure model Scrum should serve as a starting point for AMDM, since it is best known in practice (cf. Fig. 29).

Finally, the concept of architecture is considered under the aspect of agile development (cf. Chapter 5.6), because architecture is an essential building block in model-driven development.

These findings and definitions from chapters 2 to 5 now form the essential basis for the further definition of the Agile Model-Driven Method (AMDM).

In the following chapter, the necessary definitions of terms are made before the essential concepts and their implementation in AMDM are described.

6.3 Definitions

For a better understanding, the following terms are defined from the context of agile and model-driven development and their meaning in the context of AMDM.

6.3.1 Team and Role

The team includes all employees who are involved in the development of the application system. Depending on their qualifications, they assume a role in the development process and thus take responsibility for creating and editing artefacts (documents, models, source code).

6.3.2 Backlog

Backlog is a term from the agile Scrum process model (cf. [28] and [85])) and describes an ordered listing of the requirements for the software to be created. The requirements can be functional and non-functional. During the development of the project, the backlog entries are continuously reduced. Backlog entries are grouped, prioritized and estimated

for their workload. A backlog is usually broken down into smaller groups of backlog entries, which are then processed together in an iteration.

6.3.3 Iteration

An iteration describes a repetition of similar or comparable work with the aim of solving a problem step by step. The iteration is limited in time and content. In the agile procedure model Scrum, an iteration is called a sprint. AMDM distinguishes between iterations in Initial Sprint, Domain Sprint and Value Sprint.

6.3.4 Architecture

6.3.4.1 Software Architecture

The software architecture describes the basic structure of a software system, its components and their properties and interdependencies [23][89]. Definitions of the software architecture refer to the entire software system and define the basic design of the software. During the definition of the software architecture, decisions are made regarding the technology used, such as programming languages, frameworks, databases, etc. Non-functional requirements are usually the basis for decision-making for structure and technology decisions.

6.3.4.2 Domain Architecture

In the context of model-driven software development, Domain Architecture (cf. [91] and chapter 2.2.3) means all the artefacts needed to describe a domain-specific language and convert it into source code or other artefacts. This includes, in particular, the metamodel with the definition of the language elements, the transformation rules for generating the source code and the programming model into which the transformation is to be carried

out. The latter is derived from the selected application architecture. In a broader sense, the Domain Architecture also includes tools for transformation, model and DSL editors, reference models and reference implementations.

6.3.5 Domain-specific Language (DSL) and Metamodel

A domain-specific language is a modeling language that describes the properties of a particular problem domain. UML-based domain-specific languages extend the language scope of the UML modeling language by providing the necessary language properties for the problem domain. Basis is the metamodel of UML. Extensions of UML to domain-specific language elements are referred to as UML profiles in accordance with UML.

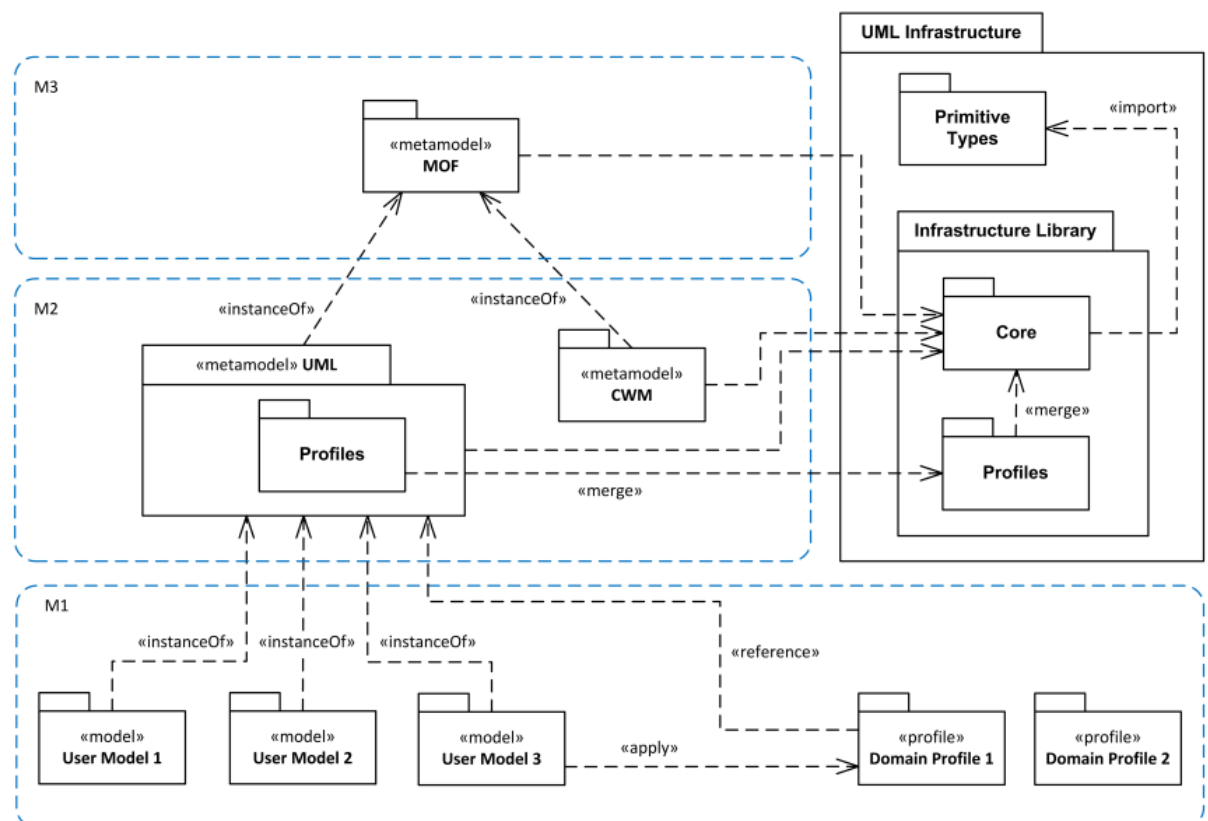


Fig. 30: UML, Meta-meta-Models and Profiles⁴

⁴ <http://www.uml-diagrams.org/uml-meta-models.html> (checked 15/8/2015)

6.3.6 Transformation and Transformation Rules

In model-driven software development, transformation always takes place on the basis of a metamodel and transfers a source model to a target model. Transformations can be model-to-model transformations (M2M) or model-to-code transformations (M2C). Transformation rules describe the type of transformation and are always based on the constructs of the domain-specific language (defined by the corresponding metamodel). Model-to-code transformations are usually performed by code generators.

6.4 Basic Concepts

The terms defined in the previous chapter have to be interpreted and defined for the Agile Model-Driven Method. In addition, there are different working techniques from agile action models such as Scrum, XP and others that have to be transferred into this context. An example of this is the role of the Product Owner in Scrum and its significance for the Agile Model-Driven Method.

6.4.1 Teamwork

Teamwork and communication are at the forefront of all agile action models. At AMDM, too, the team as a group of people has a task to fulfil together. The team is interdisciplinary. Each team member has its own know-how and contributes to the success of the project. In AMDM, the team is divided into three main groups:

- The first group knows and understands the functional requirements of the business application. On the one hand, there is the typical Product Owner, who represents the customer's point of view in the project, identifies and prioritizes

the requirements. In addition, there are also project staff members who model the problems using the domain-specific language.

- The second group defines the architecture of the application and the Domain Architecture for model-driven development. It defines the domain-specific language formally using metamodeling and creates the necessary rules for model-to-code transformations.
- The third group is the group of Application Developers who manually add non-generated functionality to the generated source code according to the architecture specifications.

If you are a larger team, it is advisable to create an intermediary between the groups to promote and control communication. In [34], Eckstein described how to deal with large and distributed teams and described this role as a possible improvement in communication.

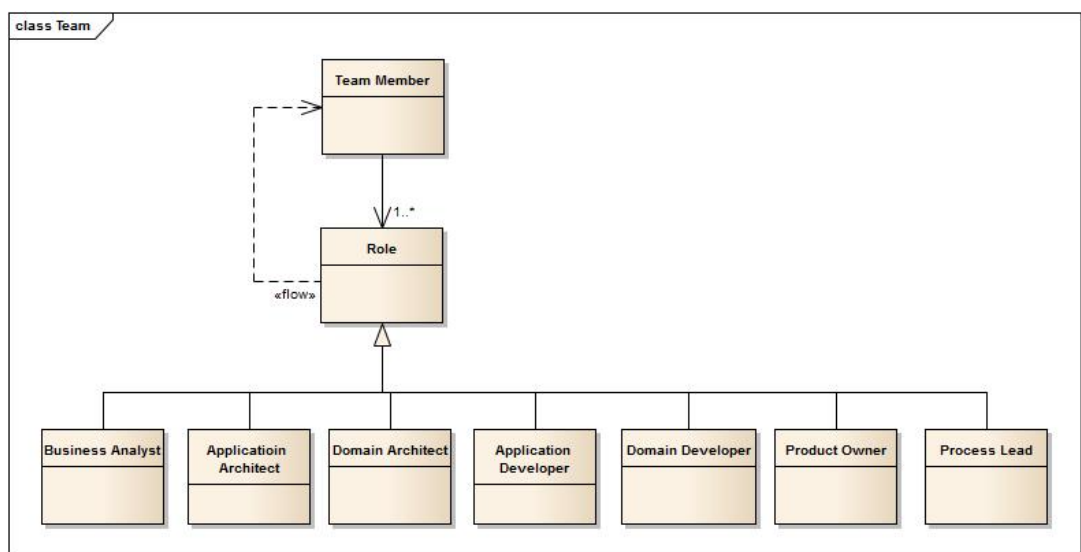


Fig. 31: Team Structure in AMDM

An important task for the team is to establish rules for development and cooperation in the project in terms of self-organization. These rules should be simple and serve as "guidelines" for decisions or quality control. Examples of such rules are, for example, the SOLID principles⁵ of Robert C. Martin or guidelines for an emergent design of architecture.

In order to constantly improve the cooperation and the results, reviews and retrospectives must be scheduled at regular intervals. While a review is based on quality assurance of work results, the retrospective focuses on improving cooperation and process improvement. This is common in Scrum projects at the end of each sprint and is also available here[28]. With this review, the team can check compliance with its self-defined rules and quality criteria and react accordingly.

6.4.2 Evolutionary Software Architecture

The software architecture is at the center of model-driven development. ISO 42010-2011 defines software architecture as "fundamental concepts or characteristics of a system in its environment that are embodied in its elements, relationships and principles of its design and evolution". It thus defines the scope for the development of structures and connections of the individual components. In AMDM, the software architecture is developed evolutionarily. This is in contrast to the usual model-driven development, where the architecture has to be fixed in large parts at the beginning. It starts with the creation of a "big picture": It is a first structuring of the software architecture with regard to

⁵ <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod> (checked on 21/10/2016)

- the distribution of components,
- layer separation,
- the identification of services / components,
- the definition of infrastructure and used frameworks.

During the creation of the "big picture", the non-functional requirements and acceptance criteria named in the backlog are taken into account. Examples for this are:

- Scalability and performance
- Reliability
- Usability
- Portability
- Maintainability, etc.

In addition, there may also be additional restrictions that affect the software architecture (e. g. technology specifications, coding conventions, organizational forms, deadlines). Despite all these influences, the "big picture" is deliberately kept simple. The goal of the "big picture" is to give the entire team an impression of the structure of the system and thus promote an understanding of the entire architecture. Therefore, the involvement of all developers in defining this "big picture" is essential. This also corresponds to the agile principle of "architecture envisioning" (see chapter 5.4.2), in which the definition of architecture is initially based on a high level of abstraction and is discussed with the team. In addition, an initial implementation is carried out on this basis and a so-called "walking skeleton" is created. This walking skeleton is used to check the specified specifications and also serves as a template for defining model-to-code transformations. In addition, the

necessary infrastructure can be built on this basis. This shows how important it is to promote a common understanding of the chosen software architecture.

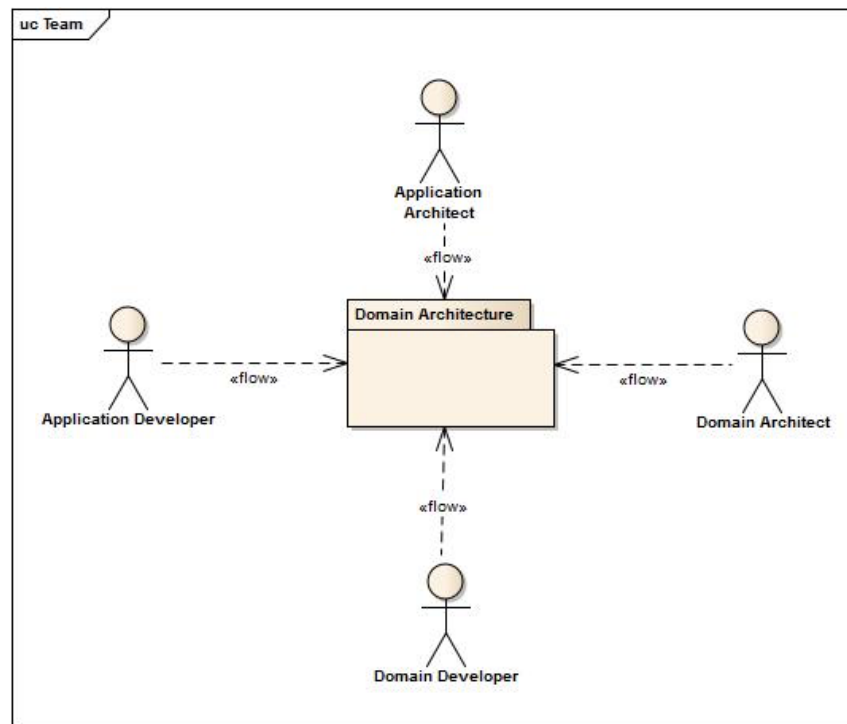


Fig. 32: Joint Development of Architecture (“Big Picture”)

Based on the coarse structure, the architecture is further refined in project progress based on feedback from development. Structural changes or code improvements are implemented. When it comes to the question of the order of architectural decisions, it is a good strategy to implement them "along" the stratification of architecture and the given dependencies. However, necessary architectural decisions should always be made at the last possible moment and therefore very late, when more knowledge and experience from development is available. As a result, sufficient architectural specifications and definitions are defined at all times in the project, thus adhering to the JBGE principle (see 5.4.4). At the same time, the rule "Assume simplicity" or "Simple design" (see 5.4.1) ensures that the architecture meets the requirements and remains comprehensible for the team.

6.4.3 Backlog Content

Compared to non-model-driven development, the non-functional content is more pronounced in the backlog of AMDM. This is due to the increasing importance of software architecture in AMDM. Basically, these non-functional requirements also influence the application in normal agile projects. In practice, however, they are often neglected in the backlog. Basic contents of the backlogs are:

- Functional requirements in the form of user stories
- Acceptance criteria and constraints (non-functional requirements)
- Non-functional requirements in the form of acceptance criteria, constraints and other quality characteristics
- Spikes⁶ (Experiments to clarify technical questions, design options and to reduce technical risks)

6.4.4 Modelling Language (DSL)

For the development of the domain-specific language, it is important to structure the problem domain first and divide it into smaller, more controllable sub-problems by partitioning. These are easier to describe because they consist of fewer elements. This has a positive effect on the complexity of the domain-specific language and its manageability.

Another good approach to structuring the problem domain is to focus on the affected objects, events, participants and locations. The domain-neutral component model according to Peter Coad et al. [24], assigns the typical elements (classes) of a problem

⁶ <http://agiledictionary.com/209/spike/> (checked on 06/10/2016)

domain to four categories (archetypes). Models that are designed based on the domain-neutral component model are similar in structure and therefore easy to understand. This assignment can also be referred to as a variant of analysis patterns (cf. [38]), which can also help to describe a problem domain.

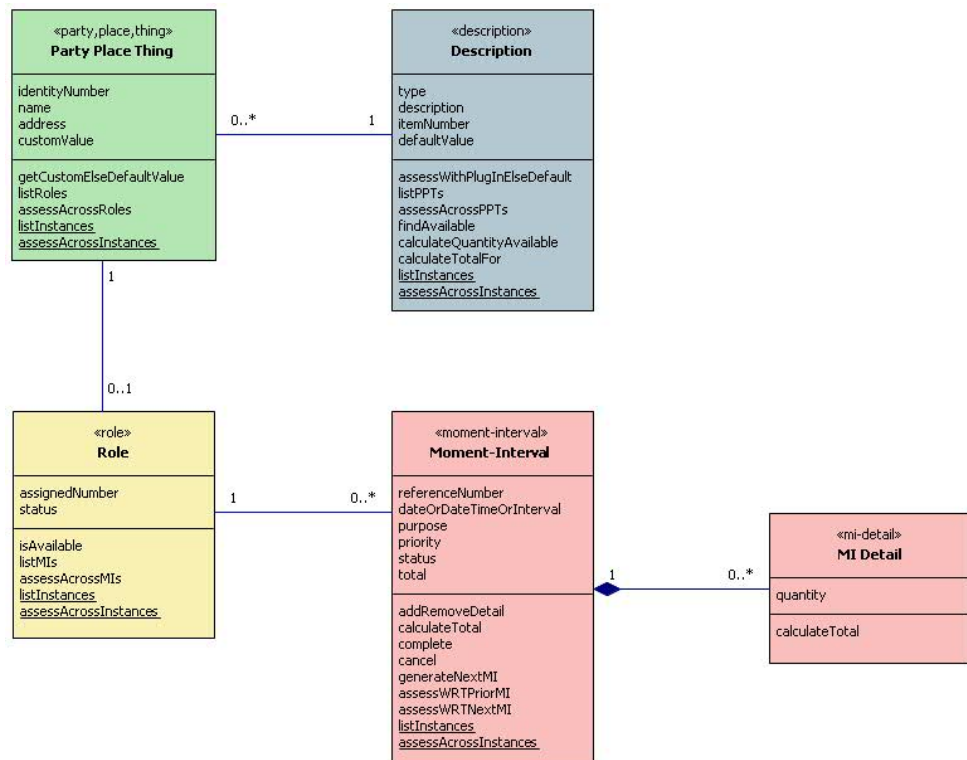


Fig. 33: Class Archetypes and Typical Associations [24]

When defining the language scope of the domain-specific language, it is important to ensure that the vocabulary of the problem domain is not mixed with technical aspects. In model-driven projects (see chapter 4) it is often observed that information for simple transformation control is also recorded as tagged values in the language. This automatically results in a dependency of the domain-specific language on these technical properties. In this way, technical and technical contents are mixed and modeled. This disturbs the design principle of the SoC (Separation of Concerns) and leads to the business

model being affected by technical changes. This leads to further problems in the further development and maintenance of DSL and transformation rules and also restricts the independent reuse of the business model.

6.4.5 Modelling in the Development Process

In existing agile process models such as Scrum, Kanban or XP, modeling is used less rather than more. Modeling is understood here more as a means of communication and understanding about the technical conditions or as a sketch for the discussion of solution concepts. This is also the opinion of Scott Ambler in [4], and the author recommends working methods such as "Model Storming" and "Iteration Modeling" (see sections 5.4.3 and 5.4.5). In Feature-Driven Development (FDD), modeling is explicitly anchored in the process, but is also understood as a means of communication in the development team, and there is also no provision for automatic processing of model information.

This is fundamentally different in the context of model-driven development and AMDM depends on the models as a basis for transformation into source code (and possibly other artefacts). Therefore, creating a business model using the domain-specific language is part of every iteration. This model is based on a user story, the corresponding source code is generated and the necessary additions are programmed. Which diagram is used depends on the facts to be described ("Multiple Models", cf. 5.4.6). If the domain-specific language does not provide the necessary language elements (e. g. diagram form or model elements), this user story and the associated model must be postponed, and the domain-specific language must be extended first.

6.5 Implementation

This chapter explains the development process of AMDM. This includes, on the one hand, the roles assumed by the team members and, on the other hand, the individual process steps and the resulting artefacts.

6.5.1 Team Members and Roles

The agile team at AMDM is divided into the group of architects, business specialists and Application Developers as described in 6.4.1. In addition, the persons responsible for controlling the development process are determined.

6.5.1.1 Product Owner

The Product Owner is borrowed from the agile process model Scrum [85] and performs the same tasks in this process. The Product Owner manages the backlog, sorts and prioritizes the entries and communicates the content to the development team. The grouping of the backlog entries follows the structure of the problem domain and is broken down by the Product Owner together with the Business Analyst into subject-related and jointly modelable units. The Product Owner determines the order in which the backlog entries are implemented, and which will be implemented together in a sprint. It determines whether further backlog entries are to be implemented functionally (see 6.5.3.3, Value Sprint) and when the Domain Architecture must be extended or adapted (see 6.5.3.2, Domain Sprint). The decision is made in consultation with the Application Architect.

6.5.1.2 Business Analyst

The Business Analyst knows the problem domain and defines the functional and non-functional requirements in the backlog. It describes these first of all via user stories, possibly anti-user stories, defines the boundary conditions and acceptance criteria. The Business Analyst works closely with the Product Owner and structures the backlog based on the structure of the problem domain and creates the individual domain backlogs (see 6.5.2.1).

In addition to formulating the requirements as text, the Business Analyst will work with Application Developers to model the requirements using the domain-specific language. In addition, the Business Analyst teaches Application Developers the necessary knowledge to manually add the necessary business logic to the generated source code

6.5.1.3 Application Architect

Based on the known non-functional requirements, boundary conditions and acceptance criteria, the Application Architect defines the appropriate software architecture. Together with the Application Developers, the Application Architect defines the application architecture and uses it to create a first minimal prototype - the "walking skeleton". The Application Architect regularly reviews the result together with the Application Developers in order to recognize the necessary changes to the architecture from the experiences during development. Together with the Domain Developers this person develops the transformation of the models into the source code.

6.5.1.4 Application Developer

The task of the Application Developers is the first implementation of the architecture as a walking skeleton. And then, above all, the manual implementation of those functional requirements that cannot be created automatically by the transformation / generation. To this end, they work closely with Business Analysts with whom they also create the models together.

6.5.1.5 Domain Architect

The Domain Architect defines the domain-specific language. For this purpose, this person develops the necessary metamodel based on the metamodels of the selected diagram types. The Domain Architect receives information from the Business Analyst who knows the problem domain and the terms used to define the domain-specific language.

6.5.1.6 Domain Developer

The Domain Developer is able to describe the transformations based on the metamodels and the software architecture. The Domain Developer defines the transformation rules for transferring business models into the source code. The Domain Developer works together with the Domain Architect and the Application Architect.

6.5.1.7 Process Lead

As compared to the Scrum Master (see [85]), the Process Lead is responsible for the compliance with the process. It ensures that the communication between the individual team members functions. For this purpose, the Process Lead calls up daily stand-up meetings and is responsible for carrying out reviews and retrospectives at the end of a

sprint. At the same time, this person represents the team against the customer and the management.

6.5.2 Artefacts

6.5.2.1 Backlog, Domain Backlog, Sprint Backlog

In AMDM there are three types of backlog, each of which is a subset of a parent backlog. The starting point is the overall backlog, which contains all requirements for the software to be created. Possible entries for the backlog are described in section 6.4.3. The Product Owner is responsible for the content of the backlogs, and decides whether new entries are added to the backlog and whether existing entries are adjusted.

The domain backlog represents a subset of the backlogs and summarizes all entries belonging to a related subset of the problem domain. An example of this is the shopping cart (sub-area) in the context of a web shop (problem domain). Another part of the same problem domain would be the articles in the assortment, for example. All backlog entries that can be assigned to such a subject-specific subarea are summarized in a domain backlog.

The sprint backlog in itself largely corresponds to the comparable counterpart in Scrum (cf. [85]) and contains all requirements to be implemented in the context of a value sprint (see 6.5.3.3). The difference to the sprint backlog at Scrum is that the requirements in the sprint backlog for AMDM originate from a domain backlog.

6.5.2.2 Software Architecture

The software architecture is developed evolutionarily in AMDM. Initially, it is designed as a "big picture" taking into account non-functional requirements, boundary conditions and acceptance criteria. This initial architecture is implemented in the form of a first minimal prototype ("walking skeleton"). This helps to build up the necessary infrastructure and minimize technical risks. The "walking skeleton" is created jointly by Application Architects and developers in a team. This development in the team increases the understanding and acceptance of the software architecture and leads to a common understanding of the structure of the future system. At the same time, the designed architecture is a prerequisite for Domain Developers to be able to define transformation rules.

6.5.2.3 Metamodel

The metamodel describes the language means of the domain-specific language. Depending on the chosen modeling language (e.g., UML) and diagram form (e.g., class diagram, activity diagram), it relies on the associated metamodels. In the case of UML, this is MOF (see [68], [71], [72]). The Business Analyst and Domain Architect as well as the Domain Developer are involved in the development of the metamodel. The Business Analyst provides the terms of the problem domain, the Domain Architect develops the corresponding metamodel and explains the relationships to the Domain Developers.

6.5.2.4 Transformation Rules

The transformation rules describe the transformation of business models into source code. These rules for M2C transformation (model-to-code) are defined by Domain Developers. This requires knowledge of the domain-specific language, its metamodel and the target

architecture of the software. In addition, this requires knowledge of the rule language used for the transformations.

The Domain Developers are supported by the Domain Architect and the Application Architect when defining the transformations.

6.5.2.5 Source Code

The source code is the target of the transformation and represents not only the pure program code but also all other artefacts that can be automated on the basis of the models (e. g. documents, configuration files).

Normally, the entire source code of an application cannot be generated. Therefore, in addition to the automated source code, there are also manually created sources. Generated and manually written source code should be strictly separated from each other as far as possible and should only be linked together by appropriate design patterns. In this case, it is advantageous if the automatically generated source text is not set under version control and is always regenerated. This avoids mixing of generated and manual code.

6.5.3 Process Steps

The main development process of the agile model-driven method is divided into three types of sprints and a parallel optimization of the software architecture. These elements are explained below.

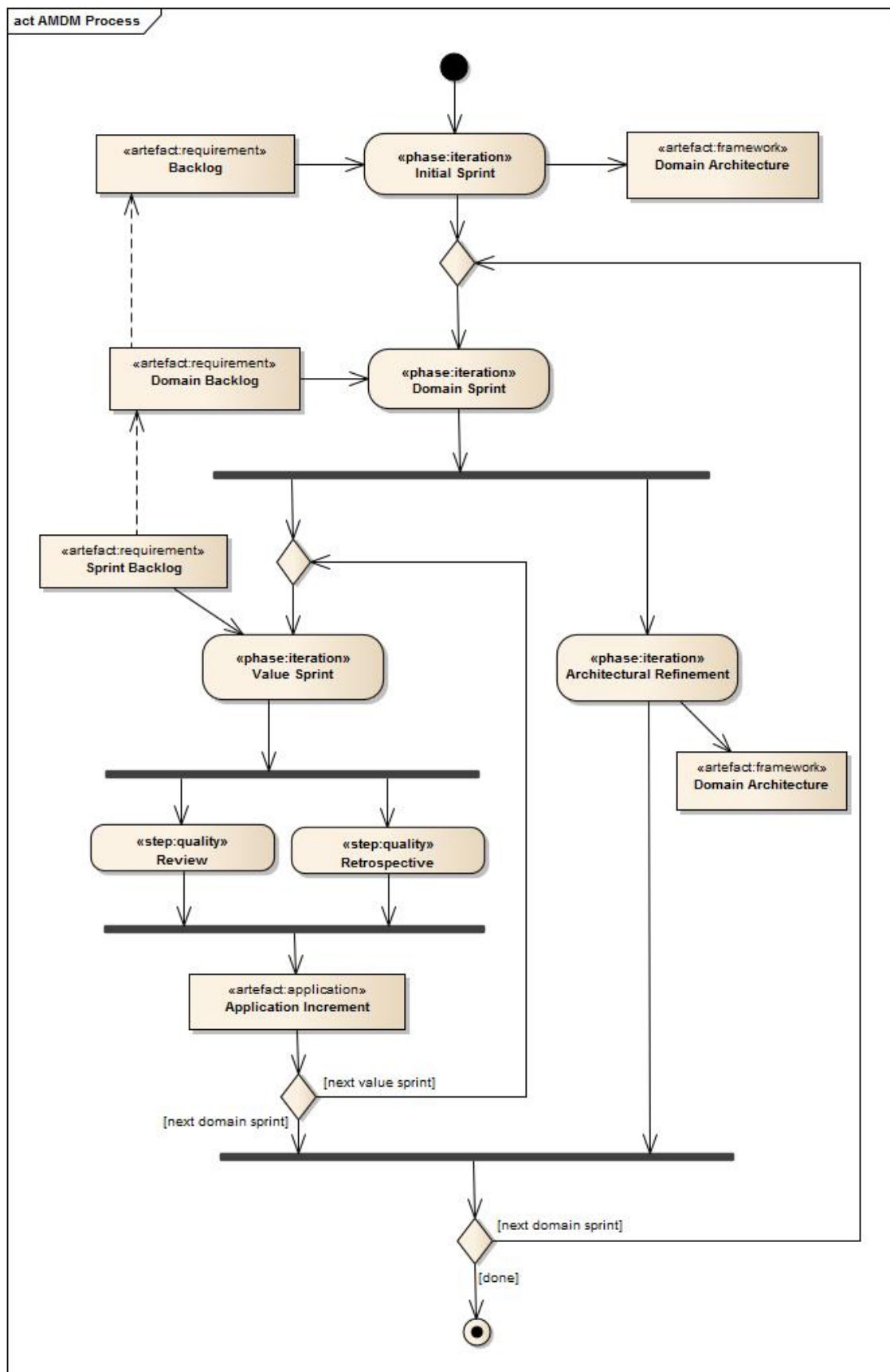


Fig. 34: AMDM Process Overview

6.5.3.1 Initial Sprint

In the initial sprint, the framework conditions and necessary foundations for the project are defined. The following activities are taking place:

- The backlogs are created by the Product Owner in cooperation with the Business Analyst. In addition, a first grouping and prioritization of the backlog entries and assignment of backlog entries to domain backlogs takes place.
- Definition of the initial software architecture in the form of a "big picture", defining the distribution, layers, services and components, required frameworks and the necessary infrastructure.
- Setting up the infrastructure. This also includes the definition of the MDSD platform, i.e. the selection of the modeling tools, the modeling language, the appropriate code generators and frameworks.
- Development of a minimal prototype ("walking skeleton")
- Team building

6.5.3.2 Domain Sprint

Within the domain sprint, the elements of the required domain-specific modeling language are defined for a specific subset of the problem domain. The requirements contained in the domain backlog determine which diagram types and elements can be used to describe these requirements. In addition, the properties of the elements of the domain-specific language are identified and defined. The domain-specific language is described using a metamodel and the Domain Developers create the necessary transformation rules for source code generation. The Domain Developers refer to the current state of the software architecture.

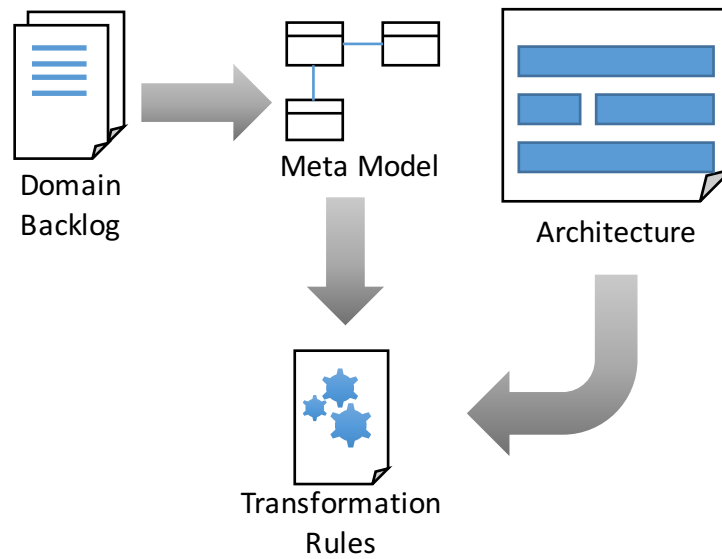


Fig. 35: Domain Sprint

The domain sprint thus provides the basis for the modeling and implementation of the requirements from the domain backlog in the following value sprints. Parallel to these activities, the Product Owner determines which requirements are implemented in the following Value Sprints.

6.5.3.3 Value Sprint

A group of requirements is implemented as part of a value sprint. This is always done in cycles. A story model is created from a user story, which describes the scenario with the help of the DSL. The defined artefacts (e. g. source code) are generated on the basis of the model and supplemented manually if necessary. These steps are carried out in short, recurring cycles in which the Business Analysts model, generate and program together with the Application Developers. This takes so long until they are convinced that the requirement has been implemented. (Definition of Done⁷).

⁷ <https://www.agilealliance.org/glossary/definition-of-done/> (checked on 30/10/2016)

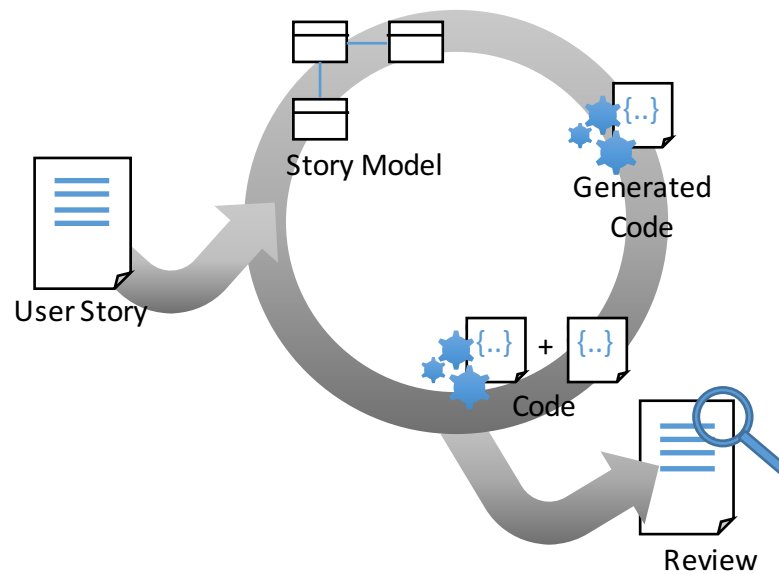


Fig. 36: Value Sprint

In AMDM, several value sprints follow each other until either all the requirements from the domain backlog are processed and implemented, or the team is of the opinion that another domain sprint is necessary. One reason for this may be that the team will notice that the domain-specific language is not sufficient and needs to be expanded or adapted to describe a situation. Another reason may be that changes to the application architecture have progressed so far that the appropriately adapted M2C transformation can be taken into account in the development.

6.5.3.4 Architectural Refinement

The software architecture is further developed in AMDM incrementally on the basis of the existing non-functional requirements. Architectural decisions are always taken at the last moment to have a more solid basis for these decisions. Additional suggestions for the adaptation and optimization of the architecture are provided by feedback from the

Application Developers from the current value sprints and from the reviews at the end of the sprints.

Matching to the changes to the software architecture, the corresponding changes must also be reproduced in the transformation rules for source code generation. If these changes have been made, they can be included in the application development. This leads to an interruption of the value sprint sequence and the execution of a domain sprint.

6.5.4 Communication

Communication has a high priority in agile process models. This is also the case in AMDM and is achieved through various feedback mechanisms as well as the close cooperation between the involved parties (for example, Business Analyst and Application Developer in a value sprint). In addition, communication is supported by regular meetings. The process lead is responsible for conducting the meetings. It also encourages regular feedback between application development and architects.

6.5.4.1 Daily Standup Meeting

In the daily standup meeting, the team is informed about the activity a team member is currently working on. The following questions are answered: What did I achieve yesterday? What do I want to achieve today? These meetings are as short as possible. They are by no means a problem solving or discussion. Problems can be addressed and addressed, but the problem solution is planned separately by the Product Owner (and possibly added as spike in the backlog).

6.5.4.2 Review

At the end of each value sprint there is a review. In this review, the results of the Value Sprint are presented and explained to the team and the Product Owner. There is also a feedback between the Application Developers and the architects. Recognized problems of the architecture can be addressed. It is also possible to check whether the specifications have also been implemented accordingly in manual coding.

There is another feedback on modeling. Here it can be judged whether the domain-specific language is correctly applied in the models or whether it is sufficiently defined for the modeling of the problem domain. The result of the review is the decision as to whether the development level reached in the value sprint is delivered or provided as an application increment.

Finally, a decision is made as to whether a further value sprint is performed or whether a domain sprint is inserted next.

6.5.4.3 Retrospective

While a review focuses on the results achieved, the process is the focus of the retrospective. The retrospective should also be carried out regularly after a value print. The retrospective will discuss whether and how the development process can be improved. Improvements can affect the team structure, the communication tools used, as well as the tools for modeling and application development. The entire team is also involved in the retrospective. However, the final decision on process changes is made by the process lead.

6.6 Summary

The Agile Model-Driven Method was introduced in the previous chapters. This development methodology is based on proven agile process elements and building blocks from existing process models for MDD. In addition, the focus is on developing the software architecture and supporting model-driven development. For this purpose, special roles for team members are defined and specific process elements are defined. An example of this is the distinction between domain sprints and value sprints and the definition of the architecture that takes place parallel to development. The special characteristics of model-driven development require close coordination between developers, Application Architects and domain specialists. This is taken into account in AMDM by the specific process elements and roles.

7 Evaluation

This chapter reviews the defined development process of the Agile Model-Driven Method. The evaluation of a process is always a difficult matter. What are the advantages of the newly defined approaches in comparison to other approaches? Is it worth switching to such a process? Measurement as an objective means of comparison does not exist. For this reason, the first step is to use the case studies described above in this chapter. The question is to what extent the use of AMDM would affect the case studies. In particular, the problems and points of criticism mentioned therein are taken into account.

In a second step, the application of the new AMDM development process will be tested on the basis of a concrete task from the author's working environment. In this way, the practical applicability of the development process will be examined.

Finally, the question is discussed to what extent AMDM is actually an agile approach for model-driven software development. Are the principles of the agile manifesto (cf. [2] and [3]) considered?

7.1 Significance for the Case Studies

As described in the introduction, the case studies described in chapter 4 will be examined and evaluated from the perspective of the new AMDM procedure.

7.1.1 Case Study 1: Interfaces to Legacy Systems

The first case study shows a problem in a very limited environment. The task was to efficiently provide interfaces to existing legacy systems in the Java world. The process of

model-driven development is very effective in this case. However, the limited task is not suitable to compare AMDM in this context.

7.1.2 Case Study 2: Software Component Development

Looking at the experiences and problems that ultimately led to the discontinuation of the model-driven development in this case study, this could have been avoided with AMDM.

The points addressed in detail:

- *The decision to develop an MDD environment came too late. There were already too many ready-made components. The prospect of payback purely in relation to the development of new components was too low.* AMDM would not have changed the late decision for the model-driven development. This was a management error. But the development of new components would have benefited AMDM more quickly from the model-driven development.
- *Too many requirements / no iterative process: It was developed too much at once. In the beginning, too many artefacts should be generated.* AMDM is an iterative and incremental development process. The Domain Architecture as the basis of the model-driven development is created successively. The waterfall-like approach to create all MDSD artefacts at the beginning would have been avoided.
- *Few intermediate results that could be used.* AMDM is designed to produce continuous and ready-to-use results. The development of the software components could have been supported much earlier.

- *The management did not trust the new technology.* By the early and steady provision of finished results, confidence in the technology could have been created. The efficient development of the software components would have been manageable and workable.

7.1.3 Case Study 3: Insurance Programming Language

The problems in the case study 3 result first from the mixing of technical and technical aspects in the domain-specific language. The resulting complexity was not mastered. The dependencies to the architecture led to a high degree of adaptation in the models and application development.

By focusing on the problem domain in the domain-specific language, AMDM avoids mixing of technical and technical elements. The architecture is developed parallel to the modeling and implementation of the technical requirements. Adaptations from architectural changes do not flow into the models, but only into the transformation. And because of the evolutionary development of the architecture, their changes are always limited with regard to their effects.

7.1.4 Other Case Studies

The following case studies from chapter 4.4 are also considered with regard to the influence of AMDM, as they also include aspects from other industrial and business sectors.

7.1.4.1 ABB Robotics and Ericsson

In his article, Staron [94] describes the experiences with model-driven development at ABB Robotics and Ericsson. The following conclusions are drawn:

- *Domain-specific languages should be designed and developed by the developers, who want to use MDD.* In this respect, AMDM takes account of the fact that the Business Analysts, architects and developers are involved in the development of the domain-specific language, the transformations and the implementation of the requirements. There is an overlap in the teaming of the teams, so that the corresponding teams always have the appropriate know-how.
- *Even excellent models do not allow complete code generation. It is not possible to dispense on manual coding.* AMDM also assumes that parts of the application must be encoded manually. This is done in tight cycles within a value sprint.
- *The MDD technology is not so far that a model-only approach like described by Brown in [18] is possible.* Whether a model-only approach will be possible depends on the quality of the DSL and the power of the transformations. It is quite conceivable that no manual coding will be necessary in some areas in the near future. The underlying process of AMDM is, however, still valid.
- *The companies are struggling with the paradigm shift from the current state of software development towards MDD and rather use proven technology instead of UML model-driven process.* By adopting concepts from known agile methods, attempts are being made to reduce the inhibition threshold compared to this new

process. Skepticism towards the model-driven technique can be countered by the rapid availability of partial results.

- *The high implementation costs of MDD can adversely affect a decision in favour of MDD.* AMDM also creates additional costs by defining the domain-specific language as well as the transformation rules. The early provision of applicable partial results places an early benefit to these costs. The quality of the automated application parts reduces future costs as part of the error analysis. In addition, the architecture is evolving evolutionarily. And this always only as far as necessary. This also avoids unnecessary costs for the creation of a bloated architecture and the resulting outlay for the M2C transformations.

7.1.4.2 Autoliv, Sectra und Saab Aerospace

The case study by Elmqvist and Nadjim-Tehrani [36] confirms, on the one hand, the saving of manually implemented code by the use of model-driven development. However, they are concerned about the availability of sufficient tools for the complete development process, from specification to implementation.

ADDM now provides a framework in which tools can be meaningfully embedded and used in a process agile and model-driven. From the requirements in the backlogs, through the evolutionary development of architecture, to problem area oriented structuring of DSL and model transformations.

7.1.4.3 IBM

IBM's case studies [22] also confirm the potential of model-driven development. However, they assume that many manual changes to models and source code will remain necessary. AMDM tries to keep the business models as stable as possible because they focus on the language of the problem domain. IBM also assumes a pure MDA approach and a two-step transformation from PIM to PSM into the code. This is not tracked in AMDM. Here a direct transformation takes place from the annotated PIM directly to the source code.

7.1.4.4 Motorola

The authors of the Motorola study [9] come to the following conclusions in their study:

- *System architects and designers tend to make implicit or explicit assumptions about the implementation of modelling.* In AMDM the intensive communication between architects, Domain Architects, Business Analysts and developers ensures that the knowledge about the models, transformations and the target architecture are evenly distributed in the team. Implicit or explicit assumptions about implementations are avoided in this way.
- *Many development teams were inflexible in changing the traditional development culture that was fostered by the absence of a defined MDD process.* To counter these reservations, AMDM relies on well-known and widely accepted agile approaches such as Scrum.
- *The third-party solutions scaled poorly and the generated code was inferior to the self-programmed solution.* Manual optimizations are always an advantage over

automatically generated source code. In AMDM, however, optimization takes place on a different level. Optimization results in an adaptation of the architecture and transformation rules. In this way, optimizations can be easily published and implemented in the entire application at the push of a button. The quality of the entire application thus increases significantly.

- *There is no development environment, which would cover all the needs of Motorola.* Whether AMDM can do this is not to be answered at this point.

7.2 Pilot Project

After the possible influence of AMDM on the investigated case studies has been presented, in a second step the concrete application of AMDM is tested by means of a pilot project. For this pilot project, a suitable problem from the author's concrete working environment was used. This evaluation was carried out between February and June 2019. The team consisted of seven working colleagues of the author, who were then asked about their experiences and impressions of AMDM.

7.2.1 Goals of the Project

The aim of the project is to provide microservices as a supplement to the existing application components of the Insurance Suite back-office solution. These application components have so far been developed as JEE multitier applications in the Java programming language. Due to their now monolithic structure, they are increasingly difficult to maintain and expand.

Now several scenarios are to be converted by the project. On the one hand, the microservices should provide a simple REST-based access to the previous complex EJB

interface of the application components. On the other hand new functionality of the specialized applications is to be made available immediately in the form of Microservices.

The following figure shows a target image from a Gartner Vendor Briefing on this topic:

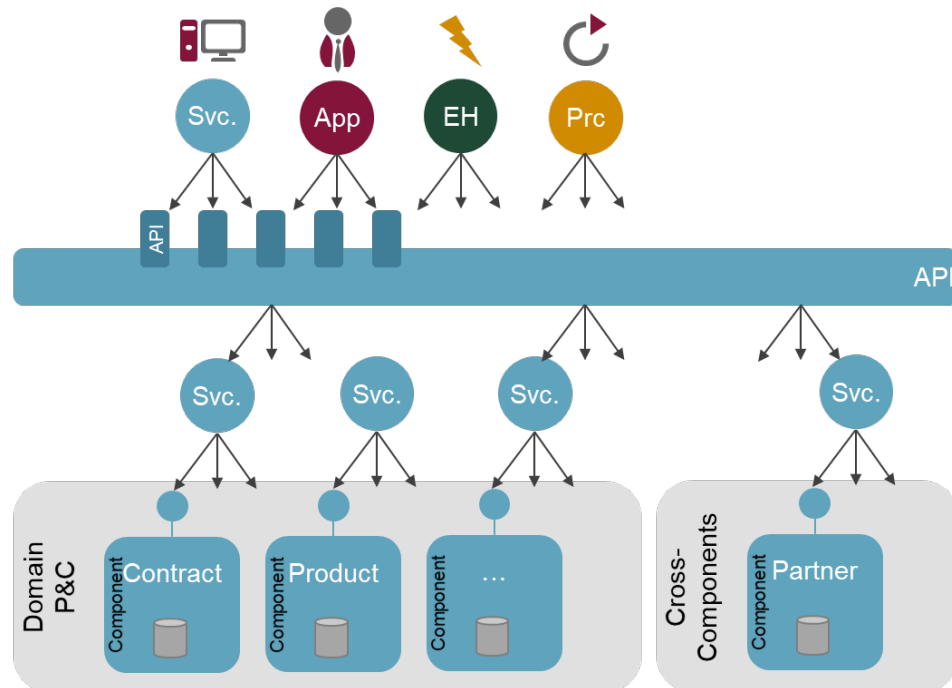


Fig. 37: Microservices in the Context of the existing Back-Office Solution

The challenge in this project was that the architecture and technical decisions about the used frameworks were not fixed at the beginning of the project. It was clear from the outset that there would be changes in the progress of the project. In addition, two types of microservices had to be developed: On the one hand, microservices, which were to act as facades for interfaces of the existing application components. On the other hand, microservices, which provide new supplementary functionality to the business applications. Both were to be kept as transparent as possible for the developers. Since it was also important to be able to easily regenerate the technical framework due to architectural adaptations, model-driven development was preferred from the outset. At the same time, the business logic should be provided early on and regularly supplemented

according to an agile approach. Thus the boundary conditions for the use of AMDM were given.

7.2.2 Team

The development team consisted of seven people and was composed as follows:

- **1 Process Lead:** This role was performed by the author of this thesis because it was about understanding and managing AMDM as a process.
- **1 Domain Architect / Domain Developer:** This role was performed by a person who already had experience in defining metamodels for UML and implementing transformation rules with the generator framework in use.
- **1 Application Architect:** This team member was particularly familiar with the architecture of the existing applications and the technical frameworks used for them. This person also designed the implementation of the microservices for the various application scenarios.
- **2 Application Developers:** These had the task of supplementing the functional logic of the new microservices and, if necessary, writing additional tests.
- **1 Business Analyst:** This team member defined the business requirements.
- **1 Product Owner:** Together with the Business Analyst, the Product Owner prioritizes the use cases (user stories) in the backlog.

7.2.3 Project Course

In coordination with all project members, a sprint length of 2 weeks was defined at the beginning. In the course of the project, a domain sprint was performed twice after the

initial sprint, followed by three value sprints each. A total effort of approx. 650 pd was required for this pilot project. In detail, the workflow was as follows:

7.2.3.1 Initial Sprint

In the Initial Sprint, the infrastructure for model-driven development was prepared. This included the specifications for the development environments, definition of the specified tools for modeling and generation or transformation. This work step was relatively simple, since corresponding tools (such as Enterprise Architect from Sparx Systems Inc. for modeling or the generator framework) had already been used and adopted in the previous environment. Thus, the first work concentrated on the development of a prototype micro service for a first use case (the contract inquiry to a property insurance contract from the inventory system). This was carried out by the Application Architect and the Application Developers. On the basis of this prototype, the first framework conditions and specifications for the software architecture were made. At the same time, the Product Owner and the Business Analyst determined in the backlog which further use cases should be developed and prioritized them together with the Process Lead for further planning.

7.2.3.2 First Domain Sprint

In the first Domain Sprint, the Domain Architect determined which UML elements should be used to describe the services (class and activity diagrams) and defined a corresponding domain-specific language for the annotation of the UML models. The basis for this was the UML model of the prototype. In addition, the necessary transformation rules were derived on the basis of this model to match the development of the prototype. The first artefacts defined were the implementation framework for the services, the API interface

with version information, the standardized call of internal services of the business components, and a basic framework for the API documentation. Subsequently, the manual development of the prototype was replaced by the generated artefacts. Thus it was clear which part of the software artefacts was generated in the first step and at which parts had to be coded manually. Thus, it was possible to explain to the Application Developers how and which parts have to be supplemented accordingly. At the same time, the Domain Architect explained to the Business Analyst the additional UML elements of the domain-specific language for creating the business models for the further services.

7.2.3.3 Three Value Sprints

In the following three value sprints, two further services were modelled by the Business Analyst in addition to a refinement of the first service. These services covered the query of the product definition for an insurance contract as well as the checking of a contract for the valid contract version at a specific point in time.

Parallel to the development, the Domain Architect and the Application Architect made further additions and adaptations to the Domain Architecture and the software architecture. Some of the work related to the exchange of used libraries (e.g. on Spring Boot) and the determination of which (partly existing) tools should be used to test the developed microservices.

At the end of each Value Sprint, a review of the created artefacts was done by the Application Architect and the Domain Architect. These provided the essential input for the adaptation and optimization of the Domain Architecture. In addition, a team meeting was held for the retrospective at which the project participants discussed how to proceed in the course of the project.

7.2.3.4 Second Domain Sprint

Due to the specifications and adaptations to the architecture regarding the tests, a second domain sprint was performed after three value sprints. In doing so, adaptations were made to the transformation regulations with regard to the architecture. In addition, new transformation rules for the generation of tests against the service interfaces or API definitions were integrated. The domain-specific language had to be extended accordingly in the metamodel so that corresponding information could be included in the business model. In addition, further modeling elements were added to differentiate between types of services. Since Domain Architect and Domain Developer were united in one person in this test project, this person was supported by the Application Developer in adapting the transformation rules. This was possible because they had the expertise from previous projects.

7.2.3.5 Three more Value Sprints

In the subsequent Value Sprints, services were implemented to report losses to property insurance, to query existing loss reports and to upload image and document data as information on the loss. These services replaced existing implementations in the existing back-office system and are used in particular in the development of the field service app for smartphones.

This test project was successfully completed with the development of these services. The generated elements of the Domain Architecture will still be used to develop further services on this basis. In the meantime, a third domain sprint has been done to make further additions in the domain-specific language and to optimize the transformation rules.

7.2.4 Project Experience

Subsequently, the participants of the project were asked about their experience with the AMDM process model. The individual interviews can be found in the appendix of this thesis.

The previous knowledge of the project members was very different. All project participants knew agile procedural models from theory, most of them also knew them from practice. There was less experience with model-driven development.

The experiences with AMDM were described as positive throughout. The essential statements are summarized:

- It is an advantage that AMDM's agile concepts are based on Scrum. By following a widespread approach, it is easier to find one's way around the process.⁸
- The combination of model-driven development and agile approach was new for all and was very positively rated.⁹
- The presentation of the procedure model in the Initial Sprint helped to understand the procedure.¹⁰
- The use of a domain-specific language for the modelling increases the expressiveness of the models. However, higher quality demands are also placed on the models, since code is generated on this basis.¹¹

⁸ See Appendix e.g. A1, A2, A4

⁹ See Appendix e.g. A2

¹⁰ See Appendix e.g. A1, A6

¹¹ See Appendix A3

- The approach was perceived as efficient because functional software was delivered in short regular cycles, teamwork worked well, and the result was completed on time. In addition, the process was found to be well structured.¹²
- The adaptation of the Domain Architecture to new technical conditions (e.g. exchange of a technical framework) during development worked well. A large initial effort could be avoided.¹³

Overall, however, the pilot project was limited in scope. Therefore, the evaluations made with regard to efficiency, project success, dependencies between artefacts, etc. are only meaningful to a limited extent. However, a larger project would have extended the time frame for an evaluation too much.

Nevertheless, it can be summarised that AMDM has succeeded as an agile process model for model-driven development in this context.

7.3 Agile Review

The fact that the Agile model-driven method supports and implements the technology of model-driven development is undisputed. But, is AMDM also an agile procedure? To clarify this, the principles from the agile manifest are used and interpreted in terms of AMDM.

As explained in chapter 3.1.2, the following four factors of the agile manifesto are the focus of agile software development:

¹² See Appendix e.g. A2, A3, A6

¹³ See Appendix e.g. A2

- The early provision of functioning software.
- Daily collaboration and personal communication between all those involved.
- The willingness and ability to always accept new customer requirements and to take them into account.
- The team organizes itself, and achieved efficiency gains.

These four points are also being pursued in AMDM. At the end of each value sprint, the delivery of a functional partial result. And the evolutionary development of architecture also supports this goal. A frequent personal communication of the team members is supported by the daily standup meetings of all participants. New requirements can be recorded at any time in the backlog and adapted in the models. Changes to the architecture and its effects on code generation are also possible at any time and are fixed in the process. And with regard to the last point, the self-organization, the retrospective mechanism is a starting point through which process and team optimization can be discussed and implemented at any time.

And what about the points that summoned Sommerville as a core statement on agility in [89] (see chapter 3.1.1)? These are:

- *Customer Involvement.* In AMDM, the customer's interests are represented by the Product Owner analogously to Scrum. This takes up new requirements, prioritizes them and leads them to the development process.
- *Incremental delivery.* This aspect has already been discussed at the outset. Frequent partial deliveries are supported.

- *People not process.* AMDM also focuses on the communication and efficient collaboration of team members. However, due to the additional complexity of the model-driven development, the process is more strongly emphasized than in other agile approaches.
- *Embrace change.* Openness towards changes is also implemented in AMDM. Functional and non-functional changes can be included in the development at any time.
- *Maintain simplicity:* The preference for simple solutions is a basic principle in the evolutionary development of software architectures. AMDM attaches great importance to adapting the complexity of the architecture to the needs of the respective requirements.

Looking at the sum of these criteria and comparing them to the Agile Model-Driven Method, this can be justly described as agile.

7.4 Summary

In this chapter, the Agile Model-Driven Method was compared with the case studies for model-driven development described above. This shows that the use of AMDM can minimize the problems and risks addressed in the case studies. The early provision of partial results increases the acceptance of model-driven development. And the evolutionary development of architecture reduces the technical risks and makes architectural principles comprehensible and acceptable for the entire team. This was also shown by the evaluation in the described test project. With regard to the development

process, AMDM can be described as agile, since the applicable principles and principles are taken into account and implemented.

8 Conclusions

8.1 Achievements

The Agile Model-Driven Method combines agile working techniques with the development approach of model-driven development. For this, the elements of model-driven development were first identified, and the existing limits and risks were considered. The criticism and skepticism of the model-driven development, which has often been expressed in practice, has also been analyzed. Case studies from specific projects in the field of the author as well as case studies from other branches of industry and business fields were used for this purpose. Thus potentials and criticisms of the model-driven development could be identified.

Agility promises to counter some of the criticisms expressed. Therefore, the first question in this thesis is whether and how modeling plays an important role in agile process models. In this case, approaches such as the MIDAS framework or the agile modeling of Scott Ambler were considered. Agile action models with a strong orientation to modeling as well as feature-driven development have also been considered. From this, it was concluded that there are individual promising approaches for an agile model-driven development, but these are not a complete solution approach.

For the definition of an agile model-driven development methodology it was necessary to characterize the project phases of an MDD project, the involved roles and artefacts. On this basis and taking into account appropriate agile modeling techniques, the Agile Model-Driven Method has been defined. It enables an agile model-driven development of business applications in a continuous process, from the specification of the requirements to the implementation. It fulfills the criteria of an agile approach and is

designed to minimize the problems and criticisms encountered in the investigated case studies.

AMDM is based on established agile process models such as Scrum. In addition to the process, AMDM also defines the involved roles and their tasks. AMDM distinguishes between technical and business content. This separation is derived from the examined procedures for model-driven development described in the literature. The relevant artefacts of domain architecture are also described and adapted in this environment.

The evaluation of AMDM was based on the investigated case studies as well as on a test project carried out under real conditions with a small team of 7 persons.

8.2 Limitations

What are the limits of the developed approach? The focus in the definition of Agile Model-Driven Method was on the development of small to medium-sized business applications, which can be built using similar software components or services. Their problem domain can be well structured, which makes it easier to break down the requirements and define the domain-specific language. In addition, the experience with agile software development is quite broad in practice in this environment. The same applies to model-driven development based on MDA or MDSD. The limits of AMDM are reached when the application requires specialized business logic or algorithms. Here, manual implementation and optimization will always be the means of choice. In this environment, therefore, both the model-driven development is also not an agile procedure. Difficult is probably a scaling to larger or distributed teams. For this, the role of a mediator is recommended.

8.3 Suggestions for Future Research

For further research, the Agile Model-Driven Method should be used in practice in other projects. The use in small and medium-sized projects as well as showcases leads to more experience and further questions regarding the development cycles and the corresponding language scope of the domain-specific language. Another open point, which can only be answered by appropriate experience, is the effort estimation and thus the planning of the sprints: How does the combination of modeling, generation and manual coding affect the effort involved in implementing a user story? Is the assumed time frame of the typical two weeks for a sprint sufficient or even too long in this case?

In addition, another field of research in connection with modelling is interesting. This concerns the quality assurance of models. How can they be validated and verified? This is an independent field of research, but the results could be interesting for AMDM to carry out reviews according to these proposals.

The classification of domain-specific languages is another field. These are no longer only used as graphical modelling languages, but increasingly also as text-based domain-specific languages. In this case, it would be helpful to examine the different types of DSL and assess their suitability for different problems.

References

- [1] Abrahamsson, Pekka (2002): Agile software development methods. Review and analysis. Espoo: VTT (VTT publications, 478).
- [2] Agile Alliance (2001): Manifesto for Agile Software Development. Available online at <http://www.agilealliance.org>, checked on 15/10/2010.
- [3] Agile Alliance (2001): Principles behind the Agile Manifesto. Available online at <http://www.agilealliance.org/principles.html>, checked on 15/10/2010.
- [4] Ambler, Scott W.; Jeffries, R. (2002): *Agile modeling. Effective practices for eXtreme programming and the Unified Process*. New York, NY: Wiley.
- [5] Ambler, Scott W. (2004): *THE OBJECT PRIMER. Agile model-driven development with UML 2.0. 3rd Edition*. New York: Cambridge University Press.
- [6] Ambler, Scott W. (2005): Feature Driven Development (FDD) and Agile Modeling. Available online at <http://www.agilemodeling.com/essays/fdd.htm>, checked on 10/10/2012.
- [7] Asadi, Mohsen; Ramsin, Raman (2008): MDA-Based Methodologies: An Analytical Survey. In: Ina Schieferdecker, Alan Hartman (Eds.): *Model Driven Architecture – Foundations and Applications*: Springer Berlin / Heidelberg (Lecture Notes in Computer Science, vol 5095), pp. 419–431.
- [8] Asadi, Mohsen; Ravakhah, Mahdy; Ramsin, Raman (2008): An MDA-Based System Development Lifecycle. In: *2nd Asia International Conference on Modeling & Simulation (AICMS)*, pp. 836–842.

- [9] Baker, Paul; Loh, Shiou; Weil, Frank (2005): Model-Driven Engineering in a Large Industrial Context - Motorola Case Study. In: *Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005*: Springer, Berlin / Heidelberg (Lecture Notes in Computer Science, vol. 3713), pp. 476 – 491.
- [10] Basili, V.R; Rombach, H.D (1991): Support for comprehensive reuse. In: *Software Engineering Journal* 6 (5), pp. 303–316.
- [11] Baskerville, R. & Pries-Heje, J. (2004): Short cycle time systems development, in: *Information Systems Journal*, 14 (3), pp. 237-264.
- [12] Beck, Kent (2002): *Test Driven Development: By Example*. Boston: Addison-Wesley.
- [13] Beck, Kent (2003): *Extreme programming explained. Embrace change. 8th print*. Boston: Addison-Wesley.
- [14] Bell, Alex E. (2004): *Death by UML Fever*. Queue, 2 (1), 2004, pp. 72-80. DOI: 10.1145/9844458.984495.
- [15] Bien, Adam (2009): *Real World Java EE Patterns - Rethinking Best Practices*. press.adam-bien.com.
- [16] Booch, Grady (2006): Blog: On Design, 03/03/2006. Available online at www.ibm.com/developerworks/mydeveloperworks/blogs/gradybooch/entry/on_design, checked on 13/09/2012.
- [17] Brambilla, Marco; Cabot, Jordi; Wimmer, Manuel (2012): *Model-Driven Software Engineering in Practice (Synthesis Lectures on Software Engineering)*. Morgan & Claypool Publishers.
- [18] Brown, Alan W.; Conallen, Jim; Tropeano, Dave (2005): Introduction: Models, Modeling, and Model-Driven Architecture (MDA). In: Sami Beydeda, Matthias Book, Volker Gruhn (Eds.): *Model-Driven Software Development. 1st ed*. Berlin, Heidelberg: Springer, pp. 1–16.

- [19] Brown, Alan W.; Conallen, Jim; Tropeano, Dave (2005): Practical Insights into Model-Driven Architecture: Lessons from the Design and Use of an MDA Toolkit. In: Sami Beydeda, Matthias Book, Volker Gruhn (Eds.): *Model-Driven Software Development. 1st ed.* Berlin, Heidelberg: Springer, pp. 403–431.
- [20] Cáceres, Paloma; Diaz, Francisco; Marcos, Esperanza (2004): Integrating an Agile Process in a Model Driven Architecture. In: *GI Jahrestagung 2004*, pp. 265–270.
- [21] Chitforoush, F.; Yazdandoost, M.; Ramsin, R. (2007): Methodology Support for the Model Driven Architecture. In: *Software Engineering Conference, 2007. APSEC 2007. 14th Asia-Pacific*, pp. 454–461.
- [22] Chowdhary, Pawan et al. (2006): Model Driven Development for Business Performance Management. In: *IBM Systems Journal (Vol. 45, No 3)*, pp. 587–605.
- [23] Clements, Paul; Felix Bachmann; Len Bass; David Garlan; James Ivers; Reed Little; Paulo Merson; Robert Nord; Judith Stafford (2010). *Documenting Software Architectures: Views and Beyond, Second Edition*. Boston: Addison-Wesley.
- [24] Coad, Peter; Lefebvre, Eric; Luca, Eric de (1999): *Java modeling in color with UML. Enterprise components and process*. Upper Saddle River, NJ: Prentice Hall PTR.
- [25] Cockburn, Alistair (2001): *Agile Software Development*. Reading, Massachusetts: Addison-Wesley.
- [26] Cockburn, Alistair (2004): *Crystal Clear: A Human-Powered Methodology for Small Teams*. Boston: Addison-Wesley.
- [27] Cohn, Mike (2004): *User Stories Applied. For Agile Software Development*. Boston: Addison-Wesley Professional.
- [28] Cohn, Mike (2010): *Succeeding with Agile. Software development using Scrum*. Upper Saddle River, NJ: Addison-Wesley.

- [29] Coram, Michael; Bohner, Shawn (2005): The Impact of Agile Methods on Software Project Management. In: *Engineering of Computer-Based Systems, 2005. ECBS '05. 12th IEEE International Conference and Workshops on the*, pp. 363–370.
- [30] Davis, Christopher W.H. (2015): *Agile Metrics in Action. How to measure and improve team performance*. New York: Manning Publications Co.
- [31] Derby, Ester; Larsen, Diana (2012): *Agile Retrospectives. Making Good Teams Great*. Dallas, Raleigh: The Pragmatic Bookshelf.
- [32] Dijkstra, Edsger W (1982): On the role of scientific thought. In: *Selected writings on Computing: A Personal Perspective*. New York, NY, USA: Springer-Verlag. pp. 60–66.
- [33] Dingsøy, Torgeir; Dybå, Tore; Moe, Nils Brede (2010): Agile Software Development: An Introduction and Overview. In: Torgeir Dingsøy, Tore Dybå, Nils Brede Moe (Eds.): *Agile Software Development. Current Research and Future Directions*. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg, pp. 1–12.
- [34] Eckstein, Jutta (2004): *Agile software development in the large. Diving into the deep*. New York: Dorset House Pub.
- [35] Eckstein, Jutta (2010): *Agile software development with distributed teams. Staying agile in a global world*. New York: Dorset House Pub.
- [36] Elmqvist, Jonas; Nadim-Tehrani, Simin (2005): Intents and Upgrades in Component-Based High-Assurance Systems. In: Sami Beydeda, Matthias Book, Volker Gruhn (Eds.): *Model-Driven Software Development. 1st ed.* Berlin, Heidelberg: Springer, pp. 289–303.
- [37] Engels, G.; Hess, A.; Humm, B.; Juwig, O.; Lohmann, M.; Richter, J.-P. (2008): *Quasar Enterprise: Anwendungslandschaften serviceorientiert gestalten*. Heidelberg: Dpunkt Verlag.

- [38] Fowler, Martin (1996): *Analysis Patterns: Reusable Object Models. 1st Edition*. Boston: Addison-Wesley.
- [39] Fowler, Martin; Beck, Kent (2010): *Refactoring. Improving the design of existing code*. 24th ed. Boston: Addison-Wesley.
- [40] Frankel, David S. (2003): *Model driven architecture. Applying MDA to enterprise computing*. Indianapolis: Wiley (OMG Press).
- [41] FZI Forschungszentrum Informatik (2010): Umfrage zu Verbreitung und Einsatz modellgetriebener Softwareentwicklung. Abschlussbericht. Freiburg, FZI Forschungszentrum Informatik, Karlsruhe. Available online at <http://www.mdsd-umfrage.de/mdsd-report-2010.pdf>, checked on 03/02/2013.
- [42] Gavras, Anastasius; Belaunde, Mariano; Pires, Luís Ferreira; Almeida, João Paulo A. (2004): Towards an MDA-Based Development Methodology. In: Flavio Oquendo, Brian Warboys, Ron Morrison (Eds.): *Software Architecture*: Springer Berlin / Heidelberg (Lecture Notes in Computer Science, vol. 3047), pp. 230–240.
- [43] Gervais, Marie-Pierre (2002): Towards an MDA-Oriented Methodology. In: *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*. Washington, DC, USA: IEEE Computer Society (COMPSAC '02), pp. 265 - 270.
- [44] Gervais, Marie-Pierre (2003): ODAC: An Agent-Oriented Methodology Based on ODP. In: *Autonomous Agents and Multi-Agent Systems 7*, pp. 199–228.
- [45] Giang, Vivian (2013): *The 'Two Pizza Rule' Is Jeff Bezos' Secret To Productive Meetings*. Available online at <https://www.businessinsider.com/jeff-bezos-two-pizza-rule-for-productive-meetings-2013-10?IR=T>, checked on 2019/05/18.
- [46] Guelfi, Nicolas; Razavi, Reza; Romanovsky, Alexander; Vandenberg, Sébastien (2004): DRIP Catalyst: An MDE/MDA Method for Fault-tolerant Distributed

- Software Families Development. In: *OOPSLA and GPCE Workshop on Best Practices for Model Driven Software Development*.
- [47] Hailpern, B.; Tarr, P. (2006): Model-driven development: the good, the bad, and the ugly. In: *IBM Systems Journal 45*, pp. 451 - 461.
- [48] Hammarberg, Marcus; Sundén, Joakim (2014): *Kanban in Action*. Shelter Island: Manning Publications.
- [49] Heijstek, Werner; Chaudron, Michel R.V (2010): The Impact of Model Driven Development on the Software Architecture Process. In: *Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on*, pp. 333–341.
- [50] Highsmith, J. (2000): *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. New York: Dorset House.
- [51] Highsmith, J.; Cockburn, A. (2001): Agile software development: the business of innovation. In: *Computer 34 (9)*, pp. 120–127.
- [52] Hildebrand, Tobias; Korthaus, Axel (2004): A Model-Driven Approach to Business Software Engineering. In: *Proceedings of the 8th World Multi-Conference on Systemics, Cybernetics and Informatics*. Orlando, USA, pp. 74–79.
- [53] Hummel, Oliver; Atkinson, Colin (2007): Supporting Agile Reuse Through Extreme Harvesting. In: Giulio Concas, Ernesto Damiani, Marco Scotto, Giancarlo Succi (Eds.): *Agile Processes in Software Engineering and Extreme Programming*: Springer Berlin / Heidelberg (Lecture Notes in Computer Science, vol. 4536), pp. 28–37.
- [54] Hunt, John (2006): *Agile Software Construction*. London: Springer.
- [55] IEEE Standard Glossary of Software Engineering Terminology (1990). In: *IEEE Std 610.12-1990*, p. 1.

- [56] ISO/IEC Standard for Systems and Software Engineering - Recommended Practice for Architectural Description of Software-Intensive Systems (2007). In: *ISO/IEC 42010 IEEE Std 1471-2000 First edition 2007-07-15*, pp. c1 -24.
- [57] ISO, IS 10746-x (1995): ODP Reference Model Part x.
- [58] Jacobson, Ivar; Booch, Grady; Rumbaugh, James (1999): *The Unified Software Development Process*. Reading: Addison Wesley Publishing.
- [59] Jones, C. (2008): *Applied Software Measurement: Global Analysis of Productivity and Quality 3rd ed.*, New York: Mcgraw-Hill Professional
- [60] Kent, Stuart (2002): Model Driven Engineering. In: *Proceedings of the 3rd International Conference on Integrated Formal Methods*. London, UK, UK: Springer (IFM '02), pp. 286 - 298.
- [61] Kleppe, Anneke; Warmer, Jos; Bast, Wim (2003): *MDA explained. The model driven architecture: practice and promise*. Boston: Addison-Wesley.
- [62] Komus, Ayelt (2014): *Status Quo Agile 2014. Study on success and forms of usage of agile methods*. University of Applied Sciences Koblenz. Available online at <http://www.hs-koblenz.de/en/rmc/fachbereiche/wirtschaft/forschung-projekte-weiterbildung/forschungsprojekte/status-quo-agile-en/>, checked on 10/09/2016
- [63] Larman, C. (2002): *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process*. Englewood Cliffs, NJ: Prentice Hall.
- [64] Larrucea, Xabier; Diez, Ana belen Garcia; Mansell, Jason Xabier (2004): Practical Model Driven Development Process. In: D. H. Akehurst (Ed.): *Second European Workshop on Model Driven Architecture (MDA) with an emphasis on Methodologies and Transformations*. Canterbury, UK, 7th-8th September 2004. University of Kent, pp. 99–108.

- [65] Lindvall, M.; Muthig, D.; Dagnino, A.; Wallin, C.; Stupperich, M.; Kiefer, D. et al. (2004): Agile Software Development in Large Organizations. In: *Computer* 37 (12), pp. 26–34.
- [66] Martin, Robert C. (2003): *Agile Software Development, Principles, Patterns, and Practices*. Upper Saddle River: Prentice Hall.
- [67] Meyer, Bertrand (2014): *Agile! The Good, the Hype and the Ugly*. Zurich: Springer International Publishing Switzerland.
- [68] Miller, Joaquin; Mukerji, Jishnu (Ed.) (2003): MDA Guide Version 1.0.1. Object Management Group (OMG). Available online at <http://www.omg.org/cgi-bin/doc?omg/03-06-01>, checked on 16/04/2011.
- [69] Mohagheghi, Parastoo; Dehlen, Vegard (2008): Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In: Ina Schieferdecker, Alan Hartman (Eds.): *Model Driven Architecture – Foundations and Applications*: Springer Berlin / Heidelberg (Lecture Notes in Computer Science, vol. 5095), pp. 432–443.
- [70] Nandhakumar, J. and Avison J. (1999): The fiction of methodological development: a field study of information systems development. In: *Information Technology & People* 12 (2), pp. 176–191.
- [71] Object Management Group (OMG) (2011): Meta-Object Facility (MOF) Specification, Version 2.4.1 (August 2011). Available online at <http://www.omg.org/spec/MOF/2.4.1>, checked on 20/10/2011.
- [72] Object Management Group (OMG) (2001): UML Profile for Enterprise Distributed Object Computing. Document ptc/2001-12-04.
- [73] Palmer, S.R.; Felsing, J.M. (2002): *A Practical Guide to Feature-Driven Development*. Englewood Cliffs, NJ: Prentice Hall.

- [74] Parsons, David; Ryu, Hokyoung; Lal, Ramesh (2007): The Impact of Methods and Techniques on Outcomes from Agile Software Development Projects. In: Tom McMaster, David Wastell, Elaine Ferneley, Janice DeGross (Eds.): *Organizational Dynamics of Technology-Based Innovation: Diversifying the Research Agenda*, vol. 235: Springer Boston (IFIP International Federation for Information Processing), pp. 235–249.
- [75] Parviainen, Päivi; Takalo, Juha; Teppola, Susanna; Tihinen, Maarit (2009): Model-Driven Development. Processes and practices. Available online at <http://www.vtt.fi/inf/pdf/workingpapers/2009/W114.pdf>.
- [76] Patton, Jeff; Economy, Peter (2014): *User Story Mapping*. Sebastopol: O’Reilly.
- [77] Pei-Breivold, Hongyu; Sundmark, Daniel; Wallin, Peter; Larsson, Stig (2010): What Does Research Say About Agile and Architecture? In: *The Fifth International Conference on Software Engineering Advances (ICSEA 2010)*: IARIA.
- [78] PENTASYS (2012): Agile Softwareentwicklung. Die wichtigsten Methoden. Statusreport 2012. Munich: PENTASYS AG
- [79] Putman, J.R. (2001): *Architecting with RM-ODP*. Upper Saddle River, New Jersey: Prentice Hall PTR.
- [80] Ramesh, Balasubramaniam; Cao, Lan; Mohan, Kannan; Xu, Peng (2006): Can distributed software development be agile? In: *Communications of the ACM* 49, pp. 41 - 46.
- [81] Reggio, Gianna; Leotta, Maurizio; Ricca, Filippo (2014): *Who Knows/Uses What of the UML: A Personal Opinion Survey*. In: Dingel, Juergen; Schulte, Wolfram; Ramos, Isidro; Abrahao, Silvia; Insfran, Emilio (Eds.): *Model-Driven Engineering Languages and Systems*. Proceedings of the 17th Internat. Conference MODELS 2014, 8th Internat. Conference SAM 2014: Springer (Lecture Notes in Computer Science, vol. 8767), pp. 149-165.

- [82] Rüpping, Andreas (2003): *Agile Documentation*. Chichester: John Wiley & Sons.
- [83] Scacchi, Walt (2001): Process Models in Software Engineering. In: J.J Marciniak (Ed.): *Encyclopedia of Software Engineering. 2nd Edition*. New York: John Wiley and Sons, Inc.
- [84] Schmidt, D.C (2006): Model-Driven Engineering. In: *Computer* 39 (2), pp. 25–31.
- [85] Schwaber, K. (2004): *Agile Project Management with Scrum*. Seattle: Microsoft Press.
- [86] Schwaber, K.; Beedle, M. (2002): *Agile Software Development with Scrum*. Englewood Cliffs, NJ: Prentice Hall.
- [87] Software Engineering Institute (SEI)/CarnegieMellon (2012): Architecture Tradeoff Analysis Method. Available online at <http://www.sei.cmu.edu/architecture/tools/evaluate/atam.cfm>, checked on 07/09/2012.
- [88] Singh, Yashwant; Sood, Manu (2009): Model Driven Architecture: A Perspective. In: *International Advance Computing Conference (IACC 2009)*. Patiala, India, 6-7 March 2009. IEEE.
- [89] Sommerville, Ian (2011): *Software Engineering. 9th ed.* Boston, Massachusetts: Pearson.
- [90] Soo Dong Kim; Hyun Gi Min; Jin Sun Her; Soo Ho Chang (2005): DREAM: A Practical Product Line Engineering Using Model Driven Architecture. In: *Information Technology and Applications, 2005. ICITA 2005. Third International Conference on*, vol. 1, pp. 70–75.
- [91] Stahl, Thomas; Völter, Markus; Bettin, Jorn; Czarnecki, Krzysztof; Stockfleth, Bettina von (2006): *Model-Driven Software Development. Technology, Engineering, Management*. Chichester: Wiley.

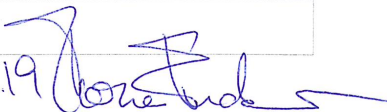
- [92] Stapleton, J. (1997): *DSDM Dynamic Systems Development Method*. Harlow, UK: Pearson Education.
- [93] Stapleton, J. (2003): *DSDM: Business Focused Development, 2nd ed.* Harlow, UK: Pearson Education.
- [94] Staron, Mirosław (2006): Adopting Model Driven Software Development in Industry - A Case Study at Two Companies. In: Oscar Nierstrasz, Jon Whittle, David Harel, Gianna Reggio (Eds.): *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006*, Genova, Italy, October 1-6, 2006, Proceedings: Springer (Lecture Notes in Computer Science, vol. 4199), pp. 57–72.
- [95] Strahringer, Susanne (1996): *Metamodellierung als Instrument des Methodenvergleichs: Eine Evaluierung am Beispiel objektorientierter Analysemethoden*. Aachen, Shaker Verlag GmbH.
- [96] Streitferdt, Detlef; Wendt, Georg; Nenninger, Philipp; Nyßen, Alexander; Lichter, Horst (2008): Model Driven Development Challenges in the Automation Domain. In: *Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International Conference*, pp. 1372–1375.
- [97] The Middleware Company (2003): Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach. Productivity Analysis. Available online at http://www.omg.org/mda/mda_files/MDA_Comparison-TMC_final.pdf, checked on 10/11/2011.
- [98] Turk, Daniel; France, Robert; Rumpe, Bernhard (2002): Limitations of Agile Software Processes. In: *Third International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP 2002)*, pp. 43–46.

- [99] Turk, Daniel; France, Robert; Rumpe, Bernhard (2005): Assumptions Underlying Agile Software-Development Processes. In: *Journal of Database Management* 16 (4), pp. 62–87.

Appendix: Interviews from the Pilot Project

A1: Interview Product Owner

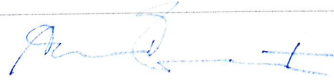
Interview: Experiences with AMDD Name: T.E. Date: July 2 nd , 2019	
1. What was your function within the project?	I was the Product Owner. In this function I have defined which services are required and in which order they should be developed.
2. Have you already had experience with agile methods before this project? If so, which methods did you use before?	Yes, I have already worked in other projects in the role of Product Owner and have specified requirements. In these projects Scrum was the used methodology.
3. Did you already have experience with the concepts of model-driven software development?	I know model-driven development only from theory. I know that parts of our software are already generated. To what extent this is based on models created in UML is beyond my knowledge. I believe that it is mostly text-based descriptions and configurations that are used as a starting point for code generation.
4. Have you found your way around the AMDD development process?	The AMDD process model was presented to us at a kick-off meeting. Since my role was similar to my previous projects, there were no major changes for me. I also didn't have to deal with process details.
5. What experience have you made with AMDD?	I worked very closely with the business analyst. I was convinced by the model-centric way of working, i.e. the description of the business logic via the story models. Especially since I've experienced how quickly executable code is created from this after we've described the user story using the models.
6. What improvements do you see in the AMDD development process?	I can't say much about that.
7. What were the hurdles that had to be overcome in the project? How did AMDD help?	I'm not aware of any difficulties.
8. How often have artifacts been delivered by you/your subteam? Which artifacts were delivered and to whom?	I was primarily busy creating the backlog. I also helped the business analyst create the user stories and story models. The backlog was the starting point for the development and was not changed so often in this small project.
9. How do you rate AMDD in terms of project efficiency?	For me, this approach made a very efficient impression. As a product owner, I was rather surprised at how quickly changes were adapted to requirements in the models and executable software was generated from them. When the tests were added, I thought it was a good idea.
10. How do you assess the model-driven development in general and specifically with AMDD?	I think I have said most about this before (see point 9).

05.07.19 

A2: Interview Domain Architect/Domain Developer

Interview: Experiences with AMDD Name: A.R. Date: July 10th, 2019	
1. What was your function within the project?	Based on my previous experience I worked as a Domain Architect in this project. And because our personal resources were limited, I also took over the role of Domain Developer at the same time. I see this as a practical solution anyway.
2. Have you already had experience with agile methods before this project? If so, which methods did you use before?	Yes, I already had experience with Scrum and eXtreme Programming from previous projects. However, in these projects we didn't develop model-driven, but always coded directly and mostly in pairs.
3. Did you already have experience with the concepts of model-driven software development?	Yes, we already use model-driven development in another project and generate technical auxiliary classes and test drivers based on models. However, these models are strongly technology-related and do not reflect the business logic as much. However, from this context I know both the generator framework used to describe the model-to-code transformation and the possibilities to develop metamodels and extend the UML with the tool Enterprise Architect as well.
4. Have you found your way around the AMDD development process?	AMDD is based on well-known patterns from Scrum. Therefore I knew of course in parts the designated roles and their tasks. It was interesting for me to what extent the activities related to model-driven development were integrated into the development process. But since I know the necessary activities and terms from model-driven development, I quickly found my way around.
5. What experience have you made with AMDD?	The process worked very well in our project context. However, the development of the services was also a limited context - but perhaps just right for testing.
6. What improvements do you see in the AMDD development process?	For me, the combination of model-driven development and agile action was new. In practice, I have not yet known this in this way. The advantages of model-driven development are well known: the generation of large parts of the software on the basis of given architecture guidelines - and this with the constant quality. And the ability to quickly adapt technical changes across the application. Now the agile approaches are added to quickly deliver results for the user. I think that's good. In the other projects, we often spent a very long time developing the DSL and the transformation rules. This is now done iteratively and is integrated into the process.
7. What were the hurdles that had to be overcome in the project? How did AMDD help?	That's hard to answer. I am not really aware of any difficulties that might have arisen. But maybe that's also because we already knew the used tools very well from the other projects.

8.	How often have artifacts been delivered by you/your subteam? Which artifacts were delivered and to whom?
<p>We have defined the basic domain-specific language in the first Domain Sprint. And also the essential transformation rules for the generation of the code. In a second domain sprint, we then included DSL extensions and the additional generation of test drivers. I was also involved in the development of the first prototype. This already resulted in insights for the definition of code generation.</p>	
9.	How do you rate AMDD in terms of project efficiency?
<p>In contrast to earlier projects, in which I was dealing with model-driven development, development steps for the implementation itself were taken much earlier. I did not know this so far. Therefore, I would actually consider the procedure more efficient than the classical model-driven development.</p>	
10.	How do you assess the model-driven development in general and specifically with AMDD?
<p>I still think that model-driven development makes sense, even if it seems to have gone out of fashion at the moment. As far as AMDD is concerned, what I wrote in point 9 applies - I think the way of iterative development of the domain architecture makes a lot of sense.</p>	

12.7.19 

A3: Interview Business Analyst

Interview: Experiences with AMDD Name: P.T. Date: July 2nd, 2019	
1.	What was your function within the project?
<p>I worked as a Business Analyst in the test project. Since I have been involved in product development for a long time, I know the business requirements very well and have already described or modeled them in various contexts.</p>	
2.	Have you already had experience with agile methods before this project? If so, which methods did you use before?
<p>Yes, a little. But only the basic principles of Scrum. The agile process models are used more in our projects than in product development. That's why I have rarely contact with these methods.</p>	
3.	Did you already have experience with the concepts of model-driven software development?
<p>No, not yet.</p>	
4.	Have you found your way around the AMDD development process?
<p>The extension of UML to a domain-specific language was new to me. In the previous modeling of requirements, I often created simple UML diagrams. Most of the time, they weren't that exact. The quality with which models have to be created here, because code is generated on their basis, was new to me. At first, this meant a change for me and I found this to be a complex process compared to the previous approach.</p>	
5.	What experience have you made with AMDD?
<p>I found the close cooperation within the team particularly positive. And I was surprised how quickly results were achieved.</p>	
6.	What improvements do you see in the AMDD development process?
<p>In the meantime, I have understood that the higher modeling requirements are necessary. Based on these models, many artifacts are generated automatically. I see this as an improvement in the quality of our software development.</p>	
7.	What were the hurdles that had to be overcome in the project? How did AMDD help?
<p>As already mentioned above, the changeover in modeling was a first hurdle for me. But otherwise I can only report positive experiences.</p>	
8.	How often have artifacts been delivered by you/your subteam? Which artifacts were delivered and to whom?
<p>Working together as a team, we actually made constant changes to the models during the Value Sprints. And we tried out directly which results were generated on this foundation. At the same time, the application developers made the necessary additions. That worked surprisingly well. I was afraid beforehand that the code generation would repeatedly destroy the developers' work. But this was not the case.</p>	
9.	How do you rate AMDD in terms of project efficiency?
<p>I have found the procedure to be quite efficient. At first I could not imagine it that way. I would be interested to see how it works in a larger context.</p>	

10. How do you assess the model-driven development in general and specifically with AMDD?

Since I didn't know the model-driven development so far, I was pleasantly surprised by the result. How much AMDD has contributed to this is hard for me to estimate.

5.7.19 

A4: Interview Application Architect


<p>Interview: Experiences with AMDD Name: S.R. Date: July 2nd, 2019</p>	
1.	<p>What was your function within the project?</p> <p>I supported the project in the role of the Application Architect.</p>
2.	<p>Have you already had experience with agile methods before this project? If so, which methods did you use before?</p> <p>Yes, I already know Scrum, XP and Software Kanban as agile process models. Scrum is usually used in our projects.</p>
3.	<p>Did you already have experience with the concepts of model-driven software development?</p> <p>Yes, too. We use model-driven development to generate our database accesses via our own persistence framework. In addition, some other artifacts are generated automatically on the basis of configuration files.</p>
4.	<p>Have you found your way around the AMDD development process?</p> <p>Since I already recognized some agile concepts from Scrum, it was easy for me. And I knew, of course, that it takes a certain amount of effort to develop model-driven concepts. But my focus was on the software architecture for the services and the definition of the technical framework.</p>
5.	<p>What experience have you made with AMDD?</p> <p>I found it interesting how well the coordination between the software architecture and the domain architecture worked. Of course, the team members have a lot to do with that. But I think it's good that this coordination is consciously anchored in the process. In this way, there are always times when these technical changes can be consciously incorporated into the development process. Of course, this also applies to the other process models - in the form of refactorings. But here it has a different dimension due to the model-driven development and generation.</p>
6.	<p>What improvements do you see in the AMDD development process?</p> <p>It is an agile development process that is specifically focused on model-driven development. I didn't know any process in this form - I actually didn't know any development process for model-driven development at all.</p>
7.	<p>What were the hurdles that had to be overcome in the project? How did AMDD help?</p> <p>Basically, I believe that changes to the software architecture that affect models or code generation should be made in a very dosed manner. Large changes may be difficult to control because of their dependencies. But this was not the case in our project. I think that in such a case the so-called domain sprints would have to be done more often. So AMDD actually has a concept for solving these problems.</p>
8.	<p>How often have artifacts been delivered by you/your subteam? Which artifacts were delivered and to whom?</p> <p>At the beginning, most of the work on my part consisted of creating the prototype and defining the basic conditions. Later I helped in the development. There were also adjustments to the used frameworks.</p>

9. How do you rate AMDD in terms of project efficiency?

That's hard to say. We were able to work quickly as a team, no one had to wait for anyone, and we were finished on time. Maybe that says enough.

10. How do you assess the model-driven development in general and specifically with AMDD?

As I said in point 6, I didn't know any development process for model-driven development. Therefore, my assessment is positive. I would be interested to know if the approach works just as well in a larger project.

12.07.19 

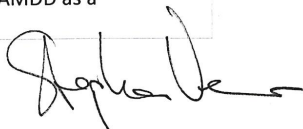
A5: Application Developer 1

Interview: Experiences with AMDD	
Name: A.W.	
Date: July 3 rd , 2019	
1. What was your function within the project?	Application Developer
2. Have you already had experience with agile methods before this project? If so, which methods did you use before?	Yes, but only to a limited extent. In the projects I was participating in, development was more "traditional".
3. Did you already have experience with the concepts of model-driven software development?	No.
4. Have you found your way around the AMDD development process?	A lot of things were new to me. At the end I was surprised by the high amount of generated code we generated. I liked the introduction to the process we got at the beginning of the project. This made it easier to classify the individual steps and also the terms.
5. What experience have you made with AMDD?	I found the reviews at the end of each sprint unusual at first. I didn't know that. But later I thought it was good. And that we talked about the process itself, what we can do differently etc.
6. What improvements do you see in the AMDD development process?	Overall, I was positively surprised by the result.
7. What were the hurdles that had to be overcome in the project? How did AMDD help?	For me a lot was new in this project. AMDD made no exception.
8. How often have artifacts been delivered by you/your subteam? Which artifacts were delivered and to whom?	At the beginning we were busy with the development of the prototype. In the further course we only added our own derived classes to the generated source code. We got most of the parts delivered, and then we produced the final product, so to speak.
9. How do you rate AMDD in terms of project efficiency?	I found the development process very good. I can't say anything about efficiency.
10. How do you assess the model-driven development in general and specifically with AMDD?	I was most surprised by the model-driven development. I didn't think we could create so many parts automatically in this project. And above all, how quickly changes could be made. That convinced me.

04.07.2019
 Aughla...
 [Handwritten signature]

A6: Application Developer 2

<p>Interview: Experiences with AMDD Name: S.W. Date: July 2nd, 2019</p>	
1. What was your function within the project?	I was one of the two Application Developers in the team for this test project.
2. Have you already had experience with agile methods before this project? If so, which methods did you use before?	I know the agile process models from my studies. Since I have only recently joined the company as a young professional, I have not yet experienced any of the process models in a real project.
3. Did you already have experience with the concepts of model-driven software development?	No. But I know the theory from my studies, too.
4. Have you found your way around the AMDD development process?	The dependence on Scrum and other agile process models was immediately apparent. What was new for me was which activities were necessary for model-driven software development. However, the process was well presented at the beginning of the project and was addressed again and again in the retrospectives.
5. What experience have you made with AMDD?	I had the impression that it was a good fit for our project size. In this respect, my experience with AMDD has been good.
6. What improvements do you see in the AMDD development process?	If I have understood it correctly, there is no agile process model for model-driven development so far. So this is certainly progress. At the university, model-driven development was presented more as a theoretical model. I was convinced that it could work this way.
7. What were the hurdles that had to be overcome in the project? How did AMDD help?	Much was new for me as a young professional. But I had good support in the team. I found the size of the team just right. I think that the short sprints and the tasks helped me a lot as well.
8. How often have artifacts been delivered by you/your subteam? Which artifacts were delivered and to whom?	We Application Developers were involved in the development of the prototype. That was then the template for the generation rules. During further development, we added the necessary manual implementations to the generated code.
9. How do you rate AMDD in terms of project efficiency?	I found the procedure well structured. And as described in theory, we regularly delivered executable software. That's why I think it was also efficient.
10. How do you assess the model-driven development in general and specifically with AMDD?	Having experienced model-driven development in practice, I am surprised that it is not used more often. I was surprised at how much we were able to generate. I can imagine AMDD as a development process very well.

5/7/19 

Publications

In English:

- Mairon, Klaus et al. (2018). Making MDD Agile: The Agile Model-Driven Method. In: Proceedings of the 5th International Conference on Computer Science and Information Technology (CSIT 2018), pp 105-118. DOI: <https://doi.org/10.5121/csit.2018.80508>
- Mairon, Klaus et al. (2010): Agile Limitations and Model-Driven Opportunities for Software Development Projects. In: Proceedings of the Sixth Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2010), University of Plymouth, UK, pp 43-52. ISBN: 978-1-84102-269-7

In German:

- Mairon, Klaus (2010): Agile und modellgetriebene Projekte. Voraussetzungen für ein erfolgreiches Zusammenspiel im verteilten Projektumfeld. In: Linssen, Oliver; Greb, Thomas; Kuhrmann, Marco; Lange, Dietmar; Höhn, Reinhard (Hg.): Integration von Vorgehensmodellen und Projektmanagement. 17. Workshop der Fachgruppe WI-VM der Gesellschaft für Informatik e.V. (GI), 1. Aufl. Aachen: Shaker (Berichte aus der Wirtschaftsinformatik), S. 122–131. ISBN 978-3-8322-9220-1, DNB: <http://d-nb.info/100407039X>

Biographical Information

Klaus Mairon, born 1967 in Freiburg, Germany, studied business informatics at the Hochschule Furtwangen University in 1994 as a Diploma in Informatics (FH). In 2005 and 2006, he completed the Master's Degree course Application Architectures, which he completed with distinction.

He works as a software architect and IT consultant at msg systems ag in the Products & Development division for the insurance industry. Since 1997 he is additionally active as lecturer for programming, architectures and object-oriented system analysis. In the beginning at the Hochschule Furtwangen University, since 2009 at the Baden-Wuerttemberg Cooperative State University in Villingen-Schwenningen. Klaus Mairon is a co-author of a textbook on object-oriented system analysis and a member of various university examination committees.

Privately, Klaus Mairon is a youth director of the Weigheim Football Club and member of the town council of his hometown. Klaus Mairon has been married since 1999 and has a son.