Model-Driven Software Modernization

Krzysztof Kowalczyk, Anna Kwiecińska
This thesis is submitted to the School of Engineering at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Computer Science. The thesis is equivalent to 24 weeks of full time studies.

Contact Information:
Authors:
Krzysztof Kowalczyk, Anna Kwiecińska
E-mail: kowalczyk.krzysztof@gmail.com, anna.ewa.kwiecinska@gmail.com

University advisors:
Bogumiła Hnatkowska, Ph.D.
Institute of Informatics
Wrocław University of Technology, Poland

Ludwik Kuźniarz, Ph.D.
School of Computing
Blekinge Institute of Technology, Sweden

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/tek
Phone : +46 457 38 50 00
Fax : + 46 457 271 25
Abstract

This thesis elaborates the Model-Driven Software Modernization (MDSM), that has been identified by us, and is defined as a group of approaches toward modernization of legacy code. MDSM approaches are based on models, tools and processes known from the Model-Driven Engineering (MDE) that aims in automation of modernization process. This thesis describes identified MDSM approaches and related standards. Additionally, it suggests that certain kinds of modernization can be implemented in a more efficient way, when a new approach, that has been proposed by the authors, is used. An exemplary modernization, that solves selected industry problem, is implemented to demonstrate that the alternative approach is feasible. In addition, the availability of tools for MDA, which can be adopted in MDSM process, is discussed and guidelines for implementing MDSM are proposed.

Keywords: modernization, migration, Model-Driven Architecture, Model-Driven Software Modernization, legacy code
# Contents

## Abstract

### 1 Introduction

1.1 Aims and objectives ........................................... 6
   1.1.1 Research questions ........................................ 7
   1.1.2 Objectives .................................................. 7

1.2 Research methodology ........................................... 7

1.3 Thesis organisation ............................................ 8

1.4 Related works .................................................. 9

### 2 System evolution ............................................. 11

2.1 Software maintenance according to ISO/IEC 14764 .................. 11
   2.1.1 Types of maintenance ....................................... 12

2.2 System evolution according to Seacord et al. ....................... 13
   2.2.1 Maintenance .................................................. 14
   2.2.2 Modernization ............................................... 15
   2.2.3 Replacement ................................................. 16

2.3 Reengineering .................................................. 17
   2.3.1 Retargeting ................................................. 18
   2.3.2 Revamping .................................................. 18
   2.3.3 Use of commercial components .............................. 19
   2.3.4 Source Code Translation .................................... 19
   2.3.5 Code reduction .............................................. 20
   2.3.6 Functional transformation .................................. 20

2.4 Used terminology .............................................. 20

### 3 Model-Driven Engineering ................................... 22

3.1 Introduction to MDE ............................................ 22

3.2 Transformations ............................................... 23
   3.2.1 Model-to-model transformations – M2M .................... 23
   3.2.2 Model-to-text transformations – M2T ..................... 25
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.3 Text-to-model transformations – T2M</td>
<td>26</td>
</tr>
<tr>
<td>4 Model-Driven Architecture</td>
<td>29</td>
</tr>
<tr>
<td>4.1 Introduction to MDA</td>
<td>29</td>
</tr>
<tr>
<td>4.2 The Four-Layer Architecture</td>
<td>30</td>
</tr>
<tr>
<td>4.2.1 Generic Four-Layer Architecture</td>
<td>30</td>
</tr>
<tr>
<td>4.2.2 MDA four-layer MOF-based architecture</td>
<td>31</td>
</tr>
<tr>
<td>4.3 Types of models</td>
<td>33</td>
</tr>
<tr>
<td>4.3.1 Computation Independent Model</td>
<td>33</td>
</tr>
<tr>
<td>4.3.2 Platform Independent Model</td>
<td>33</td>
</tr>
<tr>
<td>4.3.3 Platform Specific Model</td>
<td>33</td>
</tr>
<tr>
<td>4.4 The MDA technology base</td>
<td>34</td>
</tr>
<tr>
<td>4.4.1 Meta-Object Facility</td>
<td>34</td>
</tr>
<tr>
<td>4.4.2 Unified Modeling Language</td>
<td>35</td>
</tr>
<tr>
<td>4.4.3 XMI</td>
<td>35</td>
</tr>
<tr>
<td>4.5 Transformations in MDA</td>
<td>36</td>
</tr>
<tr>
<td>4.5.1 MOF Model to Text Transformation Language</td>
<td>36</td>
</tr>
<tr>
<td>4.5.2 MOF Query/View/Transformation</td>
<td>36</td>
</tr>
<tr>
<td>4.6 Architecture Driven Modernization Initiative</td>
<td>38</td>
</tr>
<tr>
<td>4.6.1 ADM goals</td>
<td>38</td>
</tr>
<tr>
<td>4.6.2 ADM process</td>
<td>39</td>
</tr>
<tr>
<td>4.6.3 ADM Standards</td>
<td>42</td>
</tr>
<tr>
<td>4.6.4 Knowledge Discovery Metamodel</td>
<td>43</td>
</tr>
<tr>
<td>4.6.5 Abstract Syntax Tree Metamodel</td>
<td>43</td>
</tr>
<tr>
<td>5 Model-Driven Software Modernization</td>
<td>44</td>
</tr>
<tr>
<td>5.1 Model-Driven Software Modernization in general</td>
<td>44</td>
</tr>
<tr>
<td>5.1.1 MDSM assumptions</td>
<td>45</td>
</tr>
<tr>
<td>5.1.2 Model-Driven Software Modernization Process</td>
<td>45</td>
</tr>
<tr>
<td>5.2 The “Reverse” MDA approach (RMA)</td>
<td>46</td>
</tr>
<tr>
<td>5.2.1 The RMA workflow</td>
<td>47</td>
</tr>
<tr>
<td>5.2.2 RMA with Essential PIM (RMA-EPIM approach)</td>
<td>48</td>
</tr>
<tr>
<td>5.2.3 Summary of the RMA</td>
<td>48</td>
</tr>
<tr>
<td>5.3 The Direct Transformation Approach (DTA)</td>
<td>49</td>
</tr>
<tr>
<td>5.3.1 Summary of the DTA</td>
<td>50</td>
</tr>
<tr>
<td>5.4 Common tool demands on the identified approaches</td>
<td>51</td>
</tr>
<tr>
<td>5.4.1 Metamodel</td>
<td>51</td>
</tr>
<tr>
<td>5.4.2 Model discoverer</td>
<td>52</td>
</tr>
<tr>
<td>5.4.3 Model manipulator</td>
<td>53</td>
</tr>
</tbody>
</table>
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4.4 Code generator</td>
<td>53</td>
</tr>
<tr>
<td>5.5 Suitability of MDSM approaches</td>
<td>53</td>
</tr>
<tr>
<td>5.5.1 RMA with EPIM</td>
<td>53</td>
</tr>
<tr>
<td>5.5.2 RMA with CPIM</td>
<td>54</td>
</tr>
<tr>
<td>5.5.3 DTA</td>
<td>54</td>
</tr>
<tr>
<td>6 Feasibility study</td>
<td>55</td>
</tr>
<tr>
<td>6.1 Introduction</td>
<td>55</td>
</tr>
<tr>
<td>6.2 Description of the exemplary modernization project</td>
<td>55</td>
</tr>
<tr>
<td>6.2.1 Description of the modernization context</td>
<td>57</td>
</tr>
<tr>
<td>6.2.2 Classification of the problem</td>
<td>58</td>
</tr>
<tr>
<td>6.3 Exemplary modernization project: the RMA</td>
<td>59</td>
</tr>
<tr>
<td>6.4 Exemplary modernization project: the DTA</td>
<td>67</td>
</tr>
<tr>
<td>6.5 Feasibility study analysis</td>
<td>73</td>
</tr>
<tr>
<td>6.5.1 Construction process</td>
<td>73</td>
</tr>
<tr>
<td>6.5.2 Reusability of components</td>
<td>74</td>
</tr>
<tr>
<td>6.5.3 Main challenge in the exemplary modernization</td>
<td>74</td>
</tr>
<tr>
<td>7 Challenges and recommendations in application of MDSM</td>
<td>76</td>
</tr>
<tr>
<td>7.1 Availability of existing solutions</td>
<td>76</td>
</tr>
<tr>
<td>7.1.1 MDA standards useful in MDSM</td>
<td>76</td>
</tr>
<tr>
<td>7.1.2 Availability of existing solutions</td>
<td>76</td>
</tr>
<tr>
<td>7.2 Challenges</td>
<td>78</td>
</tr>
<tr>
<td>7.2.1 Problem with complexity of model discovery</td>
<td>80</td>
</tr>
<tr>
<td>7.3 Solutions</td>
<td>83</td>
</tr>
<tr>
<td>7.4 Recommendations</td>
<td>89</td>
</tr>
<tr>
<td>7.4.1 General recommendations</td>
<td>89</td>
</tr>
<tr>
<td>7.4.2 Recommended tools</td>
<td>89</td>
</tr>
<tr>
<td>8 Conclusions</td>
<td>91</td>
</tr>
<tr>
<td>8.1 Summary of the research</td>
<td>91</td>
</tr>
<tr>
<td>8.2 High cost of entry</td>
<td>92</td>
</tr>
<tr>
<td>8.3 Answers to the research questions</td>
<td>92</td>
</tr>
<tr>
<td>8.3.1 What approaches toward software modernization, based on model transformations exist today?</td>
<td>92</td>
</tr>
<tr>
<td>8.3.2 How can existing approaches toward MDSM be refined?</td>
<td>93</td>
</tr>
<tr>
<td>8.3.3 What kind of tools are necessary to conduct MDSM?</td>
<td>93</td>
</tr>
<tr>
<td>8.3.4 In what scope MDA standards and related tools support software modernization?</td>
<td>93</td>
</tr>
<tr>
<td>8.4 Threats to validity</td>
<td>95</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Modification Request (MR) (basis of [21])</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>Evolution of software system – classification</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>System evolution [39]</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>Different types of software reengineering</td>
<td>18</td>
</tr>
<tr>
<td>2.5</td>
<td>Relationship between the definitions of maintenance</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>A general simplified representation of model discovery process</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>Generic four-layer architecture (This figure is based on [11])</td>
<td>30</td>
</tr>
<tr>
<td>4.2</td>
<td>MDA four-layer MOF-based architecture (This figure is based on [11])</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>MDA workflow</td>
<td>32</td>
</tr>
<tr>
<td>4.4</td>
<td>Types of ADM (The picture was based on [23])</td>
<td>39</td>
</tr>
<tr>
<td>4.5</td>
<td>Technical Architecture-Driven Modernization (This figure is based on [23])</td>
<td>40</td>
</tr>
<tr>
<td>4.6</td>
<td>Application/Data Architecture-Driven Modernization (This figure is based on [23])</td>
<td>41</td>
</tr>
<tr>
<td>4.7</td>
<td>Business Architecture-Driven Modernization (This figure is based on [23])</td>
<td>41</td>
</tr>
<tr>
<td>4.8</td>
<td>ADM process (This figure is based on [23])</td>
<td>42</td>
</tr>
<tr>
<td>5.1</td>
<td>The “Reverse” MDA approach (RMA)</td>
<td>47</td>
</tr>
<tr>
<td>5.2</td>
<td>The direct transformation approach</td>
<td>49</td>
</tr>
<tr>
<td>5.3</td>
<td>Common parts of the RMA and the DTA workflows</td>
<td>51</td>
</tr>
<tr>
<td>6.1</td>
<td>Main task of planned modernization</td>
<td>56</td>
</tr>
<tr>
<td>6.2</td>
<td>Relations between Java 1.4 and Java 1.5</td>
<td>57</td>
</tr>
<tr>
<td>6.3</td>
<td>Problem classification using system evolution taxonomy by Seacord, Plakosh and Lewis</td>
<td>58</td>
</tr>
<tr>
<td>6.4</td>
<td>Modernization workflow in the first prototype</td>
<td>63</td>
</tr>
<tr>
<td>6.5</td>
<td>Modernization workflow in the DTA prototype</td>
<td>68</td>
</tr>
<tr>
<td>7.1</td>
<td>EMF metamodels interoperability</td>
<td>84</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Identification of needed migration tools for the RMA and the DTA</td>
<td>52</td>
</tr>
<tr>
<td>7.1</td>
<td>Support for MDSM tools</td>
<td>77</td>
</tr>
<tr>
<td>7.2</td>
<td>Different solution toward metamodeling</td>
<td>79</td>
</tr>
<tr>
<td>7.3</td>
<td>Tools support for model discovery</td>
<td>87</td>
</tr>
</tbody>
</table>
List of Abbreviations

ADM  Architecture-Driven Modernization
ASG  Abstract Semantic Graph
AST  Abstract Syntax Tree
ASTM Abstract Syntax Tree Metamodel
CIM  Computation Independent Model
CMOF Complete MOF
CPIM Complete PIM *
CST  Concrete Syntax Tree
CWM  Common Warehouse Metamodel
DTA  Direct Transformation Approach *
DSML Domain-Specific Modeling Language
DTD  Document Type Definition
EMF  Eclipse Modeling Framework
EMOF Essential MOF
EPIM Essential PIM *
JMI  Java Metadata Interface
JPA  Java Persistence API
KDM  Knowledge Discovery Metamodel
M2M  model-to-model (transformation or transformation technology)
M2T  model-to-text (transformation or transformation technology)
MDA  Model-Driven Architecture
MDE  Model-Driven Engineering
MDSM Model-Driven Software Modernization
MDR  Metadata Repository
MOF  Meta-Object Facility
OMG  Object Management Group
PIM  Platform Independent Model
PSM  Platform Specific Model
QVT  Query/View/Transformation
QVTO QVT Operational Mappings
QVTR  QVT Relational
RMA  “Reverse” MDA Approach
ROI  Return On Investment
SPDM Software Process Engineering Metamodel
UML  Unified Modeling Language
XMI  XML Metadata Interchange
XML  eXtensible Markup Language

*Abbreviations introduced specially for this thesis
Chapter 1

Introduction

Author: Anna Kwiecińska

Legacy system is defined as the old system, that is still in use, because of its high business value [39]. Nevertheless age of used system involves some consequences. First of all, obsolete technologies are often not evolving anymore, they do not have possibility to integrate with new technologies and in time it will be more and more difficult to find experts in these outmoded solutions. All those issues significantly impede the future maintenance. Rapid change of situation on the market and development of business processes in companies generates new requirements towards existing systems that aid those processes. To meet all these new demands, and stay fully functional and usable, the system has to be maintained constantly. Thus, modernization has became a serious issue in a software industry.

What is more, to the detriment of legacy systems, developers concentrate on creation of new frameworks, optimized for fast application production instead of thinking up solutions for revitalising old software. Developers are watching changes in market trends, so a lot of emergent technologies are suited to the current needs and well-thought-out in order to simplify building the most popular types of systems, for which the requisition is raising.

An up-to-date example of such a technology-to-market adjustment is a wide variety of tools and frameworks for web applications. The Internet is expanding in a rapid pace, what contributed to a significant increase of demand for web applications in last years. The important thing is that most of these applications do not differ significantly from each other – social network services, web-based shops, ERP systems etc. have a lot in common: similar structure, recurrent components like: user accounts, logging in, etc. In respond frameworks like Ruby on Rails [51], Seam [52], Grails [48], Django [45], etc. appeared. At the same time other
technologies which were widely used in industry (e.g. Spring Framework [54]), have been redesigned to simplify development, improve maintainability and increase extensibility.

Existence of these novel solutions may affect the decision of replacing the legacy system with a new one, instead of modernizing it. Taking into consideration a possibility of further development, simplicity and time to market, products developed with usage of new technologies seem to be more profitable for corporations. Nevertheless replacement, apart from a high cost of building a new system, entails additional costs of further modernization (corporations cannot afford to replace a whole system, every time when a request for a new functionality emerges), human resources and training (developers who know the new technologies are needed, so company has to provide appropriate training or employ already trained experts). On the other hand modernization of a legacy system is also not a cheap venture. The first question that needs to be answered before starting the modernization process is who will do this? There are two possibilities: new employees, who are experts in target technologies, but need to familiarise with the old system, or the old developers, who know the system, but probably are not proficient in novel solutions. In the first case the biggest problem is a documentation, because still to much emphasis is being placed on coding in software industry, so in many cases system documentation does not arise at all, or even if it does, its level leaves a lot to be desired. This fact impedes a process of handmade modernization, which is complicated itself, and strongly affects its cost. In the second case cost and time of additional training must be counted in. Poor documentation and architecture not designed to allow changes are some of factors which make the modernization much more difficult and raise its costs. Due to this fact, so much modernization projects are abandoned, or are not finished in a planned time and budget [39]. Anyway there is another solution for the problem of legacy systems, namely an automated modernization. It decreases to a large extension the problems with lack of documentation and personnel training. But still a need for tools and methods which would aid, shorten and simplify automated modernization exists. Thus, we have decided to engage in this topic.

1.1 Aims and objectives

Authors: Krzysztof Kowalczyk, Anna Kwicińska

The main aim of this research is to present the idea of Model-Driven Software Modernization (MDSM). MDSM is an approach toward software modernization,
CHAPTER 1. INTRODUCTION

Based on Model-Driven Architecture (MDA) [27] concepts: transformations between models and code generation. Together with a transformation to extract a model of a system that is being modernized it allow for automated modernization of application. The second aim is to conduct a feasibility study of proposed MDSM approach, by solving a specific modernization problem.

1.1.1 Research questions

1. What approaches toward software modernization, based on model transformations exist today?
2. How can existing approaches toward MDSM be refined?
3. What kind of tools are necessary to conduct MDSM?
4. In what scope MDA standards and related tools support software modernization?
5. Is full automation of modernization possible and cost-effective?
6. When MDSM approach is suitable for a project?

1.1.2 Objectives

1. To make a survey of existing work in the field of software migration and software modernization.
2. To make a survey of existing work in the field of software models and model transformations.
3. To determine which standards and tools can be useful in software migration process.
4. To propose a method of modernizing software with usage of model transformation tools for selected kind of modernization problem.
5. To show feasibility of proposed approach by implementing MDSM solution toward exemplary modernization problem.
6. To provide guidelines and recommendations for usage of MDSM, basing on experience gained during the implementation of MDSM.

1.2 Research methodology

The first part of this Master’s Thesis is a literature survey in the domain related to the problem of legacy systems modernization, Model-Driven Engineering (MDE) and Model-Driven Architecture (MDA). Simultaneously, the survey and evaluation of the tools which support MDA were conducted.
Then, in the second part, the constructive research was conducted, in which basing on the theoretical body of knowledge, gathered during the survey, a model-based solution for the legacy system modernization problem, was specified. This solution is called Model Driven Software Modernization in the further parts of this thesis. Within MDSM different approaches were specified.

In the following step, in a qualitative research, the characteristics of each approach were discussed, pros and cons were identified and recommendations for what kinds of projects those approaches are suitable. Moreover, the approaches were compared in a qualitative way.

In the practical part of this thesis we tried to implement the proposed MDSM approaches. The aim of this part was to answer the question if it is possible at the moment, with the current level of tools maturity, to bring into effect the proposed modernization method, based on models and model transformations. It was presented on a small but representative project which was modernized using the identified MDSM methods. Results of the this part gave us the knowledge about the suitability of tools in this kind of modernization projects and let us collect guidelines how to put MDSM into practice, for those, who would like to use this method. The results were also the basis for further considerations and conclusions in MDSM.

1.3 Thesis organisation

- Chapter 1 Introduction – includes introduction, aims of our thesis and research questions.
- Chapter 2 System evolution and software maintenance – includes description of different approaches toward system evolution and maintenance. Moreover definitions from modernization and maintenance domain are presented.
- Chapter 3 Model-Driven Engineering – includes a description on basic concepts of Model-Driven Engineering and introduces different types of model transformations
- Chapter 4 Model-Driven Architecture – includes a description of MDA initiative and ADM project.
- Chapter 5 Model-Driven Software Modernization – includes a description of MDSM concept introduced by us, its principles and general process. There are also presented two MSMD-based approaches.
• Chapter 6 Feasibility study – includes a description of a practical part of this thesis, in which an exemplary modernization project was conducted.

• Chapter 7 Challenges and recommendations in application of MDSM – includes “a lesson learnt” from the modernization project. It contains recommendations and remarks about selection of open-source tools, which support MDSM process.

1.4 Related works

Author: Anna Kwiecińska

The problem of software migration between two similar frameworks was raised in a master’s thesis by Aseem Paul S. Cheema “Struts2JSF: Framework Migration in J2EE using Framework Specific Modeling Languages”[7]. The author created a framework for migration from the older framework – Struts to the newer Java Server Faces (JSF) [70]. Both of them are based on the model-view-controller (MVC) architectural pattern [14]. There was a need to transform separately view and controller part. Migration of a controller is pretty straightforward, because both frameworks take a similar approach in this case. But JSF provides additional user interfaces components in a view part, with server-side state. Controller migration was partially automated with usage of Framework-Specific Modeling Languages (FSML) approach, designed by M. Antkiewicz and K. Czarnecki. View migration was more complicated and is precisely described in the Cheema’s thesis [7].

Publication “Model-driven engineering for software migrations in a large industrial context” [12] describes a method of model-driven modernization, developed by Sodifrance IT Modernization company and tools for aiding the migration process. Sodifrance is now one of the European leaders in IT modernization. During its over 12 years existence on IT market, it realised over 70 successful transformation projects. From 1994 it propagates model-driven solutions for software modernization in industry [53]. In [12] authors presented a model-driven process for software migration, developed at Sodifrance. Moreover the publication contains a discussion of an MDE influence on quality and costs of software modernization and presents industrial feedback about benefits of such techniques, in a large-scale market context. In support of the thesis about the MDE influence, a case study, in which a large banking system needed to be migrated from Mainframe to J2EE, was conducted. The Sodifrance model-driven migration method consists of four fundamental phases. In the first phase, a model of the system is derived from
the legacy code. This model conforms to a metamodel of the language, is was written in, so the model is lossless — it contains all the information included in the source code. The next stage in transformation from the code model to an abstract model, independent from platform-specific details. Sodifrance uses its own metamodel, called ANT, to represent these abstract models. The third step is transformation from the abstract ANT model into a model specific for the target platform. It is implemented as a transformation from the ANT metamodel to the UML-like metamodel. The last step is a target code generation from the platform-specific model, with use of template-based code generation tools, which allow for easy handmade customization of the generated code. All the model transformation and code generation tools are part of a tool suite developed by Sodifrance, called Model-In-Action (MIA).

Another related project is Architecture-Driven Modernization (ADM) [59], an Object Management Group’s (OMG’s) initiative, which aims to standardise modernization tools and metamodels. This project is described in detail in this thesis, in Section 4.6.

Reus et al. — the authors of [38] also proposed a model-driven method of software migration. In short, they parse the original application source code and build a model of its abstract syntax tree. Next, they transform the model into a pivot language, which can be mapped into UML. Parts of this process were automated using ArcStyler [63].
System evolution covers a large scope of development activities, from minor and simple ones, like adding a field in a database to reimplementing and replacing a whole system [39]. In this chapter two different views on system evolution and software maintenance are presented. First comes from ISO/IEC 14764 “Software Engineering Software Life Cycle Processes Maintenance” standard [21] and the second one from the book “Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices” [39]. Moreover, it includes some important definitions and taxonomy in maintenance and system evolution domain.

2.1 Software maintenance according to ISO/IEC 14764

Software maintenance is a part of a software engineering process. According to ISO/IEC 14764 “Software Engineering Software Life Cycle Processes Maintenance” standard [21] maintenance processes encompass all modifications in a system, which takes place after delivery. Their role is to improve performance or other attributes of a software product, or toadapt the product to a new, modified environment.

Many reasons can cause an emergence of a need to make changes in a software system. Requests for such changes are called Modification Requests (MRs) [21] and they can be categorised as corrections or enhancements, as it is depicted on Figure 2.1. A correction is connected with disclosed or potential bugs in a system so its role is to keep the system fully operational. An enhancement’s role is extending the system, to meet constantly growing business needs.
2.1.1 Types of maintenance

Types of maintenance according to ISO/IEC 14764 standard [21]:

Corrective
Refers to modifications which are made to fix errors, found in a system after installation or when it does not meet all its requirements.

Emergency
This kind of maintenance is usually connected with corrective changes. It is a makeshift solution, right after a problem emerged, in order to keep the system operational, until corrective modifications are finished.

Preventive
Refers to modifications introduced to prevent potential problems in the future, for example to increase maintainability and reliability.

Adaptive
Concerns modifications, which are necessitated by changes in system requirements. For example different hardware platform or need for integration with other applications.

Perfective
Refers to modifications introduced after delivering the system to the client. It serves to detect early hidden faults in the system, before they come out as failures. Moreover, it includes enhancements that improve software documentation or system attributes like: performance or usability, etc.
Modifications which are part of corrective or preventive maintenance, are classified as corrections, because their role is to prevent bugs in the system, or to fix existing ones. Emergency changes were placed in corrective maintenance, because such request in most cases are caused by critical errors, which have to be fixed immediately, otherwise the system might not work correctly. Actions performed within the confines of adaptive and perfective maintenance are classified as enhancements, because they influence the system expansion and development. A detailed classification of different types of maintenance actions is presented on Figure 2.1.

2.2 System evolution according to Seacord et al.

Another definition of maintenance comes from the book “Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices” [39]. According to the book’s authors, system evolution can be divided into three main categories:

- Maintenance
- Modernization
- Replacement

A detailed taxonomy of system evolution according to [39] is illustrated on Figure 2.2. Descriptions of each of these categories can be found in sections 2.2.1 – 2.2.3.

Figure 2.2: Evolution of software system – classification

The general idea of system evolution, introduced in [39] is illustrated on Figure 2.3. This chart describes how the system evolves and what activities are performed in different phases of its life cycle. All those small and big evolution activities aim to keep the system up to the current business needs. It is not easy
task, because time of implementing the enhancements is relatively big in comparison with a growth rate of the needs. Thus, in the first phase the system is built according to the requirements, but soon the business needs grow; its level is represented by the thin solid line, and functionalities provided by the system are represented by the thick lines. Maintenance is processed in short intervals to meet current demands in possibly largest extent. But the older system is, the faster it falls behind with implementation of changes. Thus, when maintenance does not come up to the expectations anymore, a modernization effort is required. It is more time-consuming and laborious than maintenance, but also functionality increase is meaningfully higher. And finally, when further maintenance and modernization will not give the expected results, the existing system should be replaced.

Nevertheless, the borders between all those phases are fluent. Enterprises have to be aware of consequences of their decision to continue maintenance, or to stop it and carry out modernization, or maybe reimplement the system. Making the right decision requires careful analysis of the legacy system value, implications of each action and height of return on investment (ROI) in each case.

2.2.1 Maintenance

According to [39] maintenance is “an incremental and iterative process in which small changes are made to a system”. It includes these kinds of modifications in which small bugs are fixed or minor functionalities which do not disturb existing
program structure, are added. Anyway not all problems of legacy software can be solved in such way; maintenance has some limitations, too:

- Limited possibility to adopt new technologies – new, better solutions are developed in a rapid pace; using them to improve a legacy system could be very profitable. Nevertheless, adopting them often requires significant changes, which are not typical for maintenance, e.g. implementing distributed architecture or big structural changes.

- Increasing cost – costs of maintenance are getting bigger with time, difficulty in finding experts in old technologies or in-house training highly influence the cost.

- Increasing business needs – if a legacy system architecture is not designed to be expandable enough, including every new functionality becomes more difficult.

2.2.2 Modernization

Modernization is more invasive kind of system evolution than maintenance. It often includes bigger changes in the system structure, adaptation of newer technologies and high functionality enhancement. In comparison with maintenance it requires more extensive modifications, but in consequence conserves wider part of the old system. Modernization should be considered when simple modifications, within the confines of maintenance, are not sufficient to meet all the constantly growing expectations, but when the legacy system has still a high business value, which should be retained. In such situation it is better to avoid much more expensive replacement.

Modernization is categorised as:

- White-Box Modernization

- Black-Box Modernization

There is also a form of modernization, introduced by Tittley and Smith in [41], called reengineering. It is orthogonal to the division above. Because of the fact, that reengenering is strongly connected with our works, thus important to us, it is described more precisely in the separate section: 2.3

White-Box Modernization
White-box modernization is carried out when source code of a legacy system is known. This kind of modernization process must be prefaced by an initial phase, called program understanding. In this phase the system is discovered from inside. To do this reverse engineering techniques are used for extracting as much information as possible from the source code: the system structure, technological principles, etc.. The purpose of this preliminary phase is to understand better how the system works. It is also possible to partially automate this preliminary phase.

After initial phase, when the source code is understood and analysed, the main part of white-box modernization can be started. Activities in this phase often include software restructuring – “the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system’s external behavior (functionality and semantics)” [8]. This procedure’s purpose is usually improvement of some system attributes like maintainability, performance, etc..

Black-Box Modernization

When source code of a legacy system is not available, black-box modernization is performed. The legacy system in a such case is treated as a black box, which takes some data as an input, then it performs some actions and in a result it returns some output. Its operation and interfaces cannot be examined otherwise than by an analysis of those inputs and outputs. In most cases black-box modernization consists in adding a layer, which hides the complexity of the old system and provides new interfaces, suited to the needs of modernized system. This concept has been defined and systematised as an adapter design pattern by the Gang-of-Four – originators of design patterns [17].

Black-box modernization involves examining the inputs and outputs of a legacy system, within its operating context, to understand what types of data the system takes as an input, and what data it returns in a result. The abilities of this method of modernization is limited in comparison with white-box modernization – not every modernization which is possible to do using white-box, is feasible using black-box modernization methods.

2.2.3 Replacement

The old system is replaced with a completely new one, built from scratch. Replacement is appropriate when the old system cannot be improved in such a range which would meet constantly growing business needs or long-term costs of modernization and maintenance would be higher than building and maintaining a new
Replacement can be enforced in two ways: using a "big-bang" approach [39], in which the whole system is installed at once, or incrementally, when the new system is installed module by module. Incremental replacement may take a form of series of white-box modernizations [39].

Before making a decision of replacing the old system, some risks should be evaluated. Firstly, personnel not be familiar with the new technologies, so they may need training, to be able to maintain the new system. And secondly, there may occur a problem connected with testing. Legacy systems are usually well tested and stable, whereas new system require detailed testing and it may take some time before all bugs will be eliminated. This can result in temporary system malfunction.

2.3 Reengineering

Reengineering is also a form of modernization, it was defined by Tittley and Smith in [41] as “the systematic transformation of an existing system into a new form to realise quality improvements in operation, system capability, functionality, performance, or evolvability at a lower cost, schedule, or risk to the customer”. They indicate that reengineering concentrates on a system improvement so it should result in a higher ROI than replacement [39]. This approach to a legacy systems improvement is neither as cheap as the software maintenance, nor as effective as replacement, nevertheless it is very often the most pragmatic one. Practical experience and observations showed that creating a new system is not so common in companies which work on lots of legacy software because of financial reasons. Reengineering is much more cost effective in such cases, furthermore its big advantage is a lower risk of project failure [39]. There are many different reengineering strategies. As it is depicted on Figure 2.4 they can take a form of:

- Retargeting
- Revamping
- Use of commercial components
- Source code translation
- Code reduction
- Functional transformation
2.3.1 Retargeting

Retargeting is “the engineering process of transforming, rehosting, or porting the existing system in a new configuration” [22]. New configuration means:

- a new hardware platform
- a new operating system
- a Computer Automated Support Environment (CASE) platform

Sometimes adopting CASE platforms does not require changes in existing system, such cases are not treated as retargeting actions. Retargeting is usually motivated by high maintenance cost of an old platform (which are much less in newer ones) and all benefits which modern solutions can bring. Thus, changes in an old system are mostly oriented towards the most effective usage of features from a new platform [39].

2.3.2 Revamping

The term of revamping is connected with user interface (UI). It is a common form of reengineering, in which only a user interface is changed. These days, with increasingly high interest in the Internet, the most popular form of revamping is adding a web interface to legacy systems. In the past a very common practice was replacing a text interface with a graphical UI. One of methods for this kind of revamping was using a screen scraping – a black-box method, in which a tool (usually an existing component) is “redirected” from a console screen into a graphical frame of web interface [39]. This method is relatively cheap and results of a modernization are well visible. Nevertheless under the new UI is still an old system,
which has not been changed, so adding new functionalities or further maintenance is still very difficult, because system extensibility did not improve.

2.3.3 Use of commercial components

The next practice in reengineering is a replacement of parts of legacy systems with existing, commercial-of-the-shelf (COTS) components. Such a reengineering project has its both advantages and disadvantages. First of all a big plus of this solution is reducing amount of code, which will require a maintenance in the future. Anyway, when calculating costs of such a venture it is important to consider additional cost, not only the price of components, but also of all needed licences, expense of extracting parts of the legacy system, which are to be replaced, amount of glue code, that is required to adopt the components to cooperate with the old system [39].

2.3.4 Source Code Translation

Another reengineering project commonly used in business is translation of code. It is “transformation of source code from one language to another or from one version of a language to another version of the same language” [22]. It may be motivated by the lack of compatibility in the source and target platform – the old language may be not available on the new platform. It is one of the most risky reengineering projects, due to relatively high realisation cost (which is often underestimated) with relation to often overestimated benefits [39]. To become more cost-effective the migration should be automated. However, this solution is not easy and if it is not done properly it may look just like the old system rewritten in the new language without elements specific for the target platform, that raise the quality of code. Such a program may look for instance like an application in COBOL, rewritten in Java – this will not affect positively the quality. Anyway migration between languages with the same programming paradigm, for example transformation of code written in one object-oriented language into another, like Java or C#, is still difficult and risky but more likely to be successfully automated, than a source code transformation between languages which differ in programming paradigms, for example from a procedural to object-oriented language (e.g. COBOL and Java). Source code translation not always means migration from one language to another. It can be also a simple upgrade into a newer version of the same language. The less difference between versions, the simpler and less risky the transformation is. Thus, migration from Java 1.2 to Java 1.4 and than to Java 1.5 would be less risky than straightforward migration from Java 1.2 to Java 1.5.
2.3.5 Code reduction

Code reduction’s principles are quite simple. It assumes that all unnecessary and redundant code from the old application should be eliminated before the migration process begins [39]. This effort is very cost-effective, because migration of every single line from an input code generates additional cost.

2.3.6 Functional transformation

Functional transformation is a form of reengineering in which program structure is modernized. For example procedural code is transformed into structural, or structural into object-oriented, application can be also divided into modules, or the data storage may be changed. Functional transformation can be divided into three categories:

Program structure improvement
In general improving flaws in program structure. Firstly, a new alternative structure has to be chosen, and then an old structure is being replaced with the new one, e.g. replacing GOTOs with structural code. It can be performed both manually and automatically.

Program modularisation
Collecting similar concepts in separate modules. It usually decreases program complexity, increases readability and extensibility. Often used to prepare the system for replacing some parts of the system with COTS components.

Data reengineering
Because most of the software systems need source of data to process also the data storage, organization or format requires modification from time to time. Data reengineering represents this kind of modernization practices.

2.4 Used terminology

As it was shown terminology from software evolution and maintenance area, used by Seacord, Plakosh and Lewis in a book [39] differs from used in ISO/IEC 14764 standard. Because of the fact that both the book and the standard are referred in this thesis it is very important to determine similarity and relationship between terms in both meanings.

First of all, ISO/IEC 14764 standard [21] does not distinguish modernization from maintenance. The term maintenance is used, but in wider scope, than in
In ISO/IEC 14764 standard [21] maintenance includes all kinds of enhancements and modifications that are made to a system, whereas Seacord et al. and Lewis [39] made a distinction as regard to size and scope of these changes, into: maintenance and modernization. As it was written it [13] “modernization starts where existing practices fail to deliver against business objectives”. Similarly like in [39] it highlights the fact that modernization is a more invasive strategy than maintenance, which is being put into practice occasionally, when typical maintenance practices like fixing minor bugs or implementing simple functionalities. Modernization actions interfere into the system structure, sometimes requires upgrading used frameworks – such modifications affect significant part of the system and is laborious, but make easier (or in some cases just make possible) to perform further maintenance works. Relationship between two definitions of maintenance, first from ISO/IEC 14764 standard [21] and the second introduced by Seacord et al. [39], is depicted on Figure 2.5. In this thesis nomenclature from the book [39] is used, because it lets to define the area of this research more precisely.
In this section the concept of Model-Driven Engineering (MDE) is outlined. MDE technologies combines the concepts of Domain-Specific Modeling Languages (DSMLs) and model transformations. Different types of transformations: model-to-model, model-to-text and text-to-model are also described in this section.

3.1 Introduction to MDE

Before we start talking about MDE, the basic term – “model” needs to be defined. There are many definitions of this term, but the most suitable for the area of this thesis is a definition from [6]:

Model

“A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system.”

A model can be represented using both graphic and text notation, or combination of them. The text may be written in a special modeling language, as well as simply in natural language.

In model-driven solutions of system development the main principle is that “everything is a model” indicating important role of models and paraphrasing object-oriented technology principle “everything is an object” popular in ’80s. Anyway those two conceptions are not appositive. The lifecycle of system composed of objects can still be presented as a chain of model transformations [5].

The idea of Model-Driven Engineering was a response to the technology announced by the Object Management Group (OMG) consortium, called Model-Driven Architecture (MDA). More details about it can be found in the following
chapter, which is devoted to MDA concept itself. It was a try of generalising its principles about using models for developing systems at different level of abstraction. So the aim of MDE is to raise the abstraction level of the system at the beginning, in program specification and then to automate the development process. The automation of software development life cycle in MDE is achieved through executable models and techniques of model transformation and code generation. Generalising, MDE technologies combine the following concepts:

- Domain-specific modeling languages (DSMLs) – designed to formalise requirements, structure and behaviour of the application within its specific domains, for example online financial services, human resources management, computer aided laser surgery and so on. DSMLs are particularly useful for developers, who can build applications using ready-made DSML’s components. To describe DSMLs, metamodels that fulfil specific requirements, are needed. Such a metamodel has to enable defining relationships between concepts in the domain. Moreover, a precise notation which specifies semantics and constraints is needed.

- Transformation engines and generators – which role is to analyse particular aspects of a model, and to generate artifacts on its basis, such as: code, different model representation, eXtensible Markup Language (XML) deployment files, and so on. The fact that those artifacts are synthesised from models helps in ensuring the consistency between requirements toward system, captured by a model and application implementation.

3.2 Transformations

Approaches toward model transformation at the top level are divided into model-to-model (M2M) and model-to-text (M2T). Model-to-text approach is usually used for code generation; sometimes also non-code artifacts, like XML documents, are created. M2T may be viewed as a specific case of M2M transformation, if only a metamodel of the target programming language (the language of generated code) is provided. Anyway, in practice usually the code is first generated as plain text, and next compiled with usage of existing tools [9].

3.2.1 Model-to-model transformations – M2M

Model-to-model transformation is a one of the key aspects of MDE. Among model-to-model types of approaches, the following subtypes are distinguished:

- direct-manipulation
• relational
• operational
• template-based
• graph-transformation-based
• structure-driven
• hybrid

Direct-manipulation approach In this approach usually a model representation is provided, moreover some API to manipulate a model is included. Implementation of direct-manipulation approach often takes form of an object-oriented framework. Such a framework provides a basic infrastructure for organising transformations (it can take a form of for example abstract classes or interfaces), however all the transformation rules have to be implemented from scratch [10], in some specific language, for instance in Java. An example of direct-manipulation M2M approach is Jamada.

Relational approach This declarative approach is based on the concept of relations. “Relation are multi-directional declarative transformation specifications” [1]. “Relations in a transformation declare constraints that must be satisfied by the elements of the candidate models” [34]. So firstly, a type of relation between the source and target model must be stated and then, it will be specified using constraints [10]. Usually this specification is non-executable (like relations in [2] approach), it means that is not able to create or alter model; instead it can check if two models are consistent. However, it is possible that a non-executable relation will be given some executable semantics, similarly to logic programming. Thus in approach presented in [2], a distinction between relations and mappings were made. First represents bi-directional, non-executable transformation specifications, and the second, transformations which are executable, unidirectional implementations of relations [10]. It often support backtracking and in contrast to direct-manipulation [3.2.1] in relational approach target elements are created implicitly.

Operational Approaches Operational approaches are similar to the direct-manipulation ones, but they are more oriented on model transformation. In this category, the most common practice is extending “ the utilised metamodeling formalism with facilities for expressing computations” [10]. For example to extend
OCL with constructions typical for imperative languages. QVT Operational is an example of a system based on this approach. Additionally, some features, such as tracing, may be included using dedicated libraries.

**Template-Based Approaches** This category is based on model templates. The definition of model templates according to [10] is as follows: “models with embedded metacode that compute the variable parts of the resulting template instances. This metacode can take a form of annotations and the most common language to write them is OCL.

**Graph-Transformation-Based Approaches** This kind of approaches in based on theory in graph transformation. They Graph-Transformation-Based approaches work on “typed, attributed, labeled graphs” [10]. This graph was designed especially for representing UML-like models.

**Structure-Driven Approaches** This category of approaches towards model transformations has basically two phases. The first phase is a creation of the hierarchical structure of the target model. And in the second one the attributes and references are set in the target model. An example of structure-driven approach is the QVT submission by Interactive Objects and Project and OptimalJ.

**Hybrid Approaches** Hybrid approaches combine various techniques from the approaches, that were described before. QVT is an example of such a hybrid approach, it consists of three different components: QVT Relation, QVT Operational, QVT Core [34].

### 3.2.2 Model-to-text transformations – M2T

M2T approaches:

- visitor-based
- template-based

**Visitor-based**

Implementations of visitor-based approaches toward M2T transformations use the visitor design pattern [17] to traverse across whole model, translating each elements into code and printing it to a text stream.

**Template-based**

Template-based approaches toward M2T are more common in industry, than visitor-based ones. A template is usually a fragment of code with metacode...
insertions to access a source model and get necessary information from it. This logic can take form of queries in a declarative language (in example: OCL [31], XPath [56]).

3.2.3 Text-to-model transformations – T2M

Text to model transformation is a process of creating a model (usually based on language metamodel) from existing code. In this thesis two terms connected with this process are distinguished: model discoverer and T2M tool.

**Model discoverer** is a tool that can create complete model from code in a particular language, so it realise text to model transformation. Exemplary process of discovering model from code is shown on Figure 3.1. This process have been based on study of existing tools. To understand the process we need to introduce three definitions (based on a draft version of the ASTM standard [33]).

**Concrete Syntax Tree** CST or Parse Tree is a direct syntax representation of code in form of directed, finite tree. Usually CST is constructed as first artifact in process parsing text and later AST is constructed from it.

**Abstract Syntax Tree** AST is a direct finite tree representation of code but elements that does not change semantic meaning are usually removed from it. AST represent higher level of abstraction than CST. Usually code can be restored from such representation with minor changes from original (formatting).

**Abstract Semantic Graph** ASG is graph representation of code that represent code precise semantic. In this step identity of named object is resolved. This change the tree to a graph. Further abstraction and removal of any parsing related artifact can occur.

General process of model discovery starts with parsing of text artifacts and construction of CST, which is latter abstracted into AST. Model is optionally transformed further into ASG. This is done by name resolving. In this process names used in code are analysed and objects that are pointed by those names are assigned, so tree structure of code is changed into a graph. In MDSM we need a representation that can be read by MDE tools. So in next step a model is constructed from previous artifacts.

Universal model discoverer should create a PSM that can represent all semantic concepts in code. It should be resolved (like ASG) and ideally it would also
handle code comments. Such model can be used for effective code generation (like AST). ASG is usually used as internal representation in compilers/interpreters, hence it usually would strip comments and potentially can change code structure (optimisation), or even use different semantic model. In such case it is better to create PSM as kind of AST with resolved names. When such model is available we can use M2M to create models on higher level.

The example of such a tool are available in MoDisco project (MOdel DISCOveryer) [73]. Those projects supports discovering Java to create models compatible with J2SE5 metamodel. C# language metamodel is also available, but as for the time of writing there is no discoverer for it. Moreover, MoDisco provides transformations from those two metamodels into more common the Knowledge Discovery Metamodel (KDM) and from KDM to the Unified Modeling Language (UML).

**Text-to-model tool** T2M tool is a tool that support creation of model discoverer. This is usually done by generating a parser like in classic parsers generator tools (e.g. ANother Tool for Language Recognition (ANTLR)[61]), but in T2M tool, the parser output a model. As for today, this is realised in industry, by a category of tools that use EBNF-like languages, to define a language that can be parsed. Those tools are often dedicated for construction of textual Domain Specific Languages. Some of them need dedicated metamodel (like EMFText and TCS),
but other can create language metamodel from its grammar (e.g. Xtext[57]).
Chapter 4

Model-Driven Architecture

Author: Anna Kwiecińska

In this chapter the basic concepts of Model-Driven Architecture (MDA) are presented. The idea of four-layer architecture in general and MDA context is explained. Moreover the descriptions of standards used in MDA: Meta-Object Facility (MOF), UML and XML Metadata Interchange (XMI) are included. The second part of this chapter is dedicated to Architecture-Driven Modernization (ADM) initiative, so its aims, concepts and related standards are described in it.

4.1 Introduction to MDA

For better understanding of the concept of Model-Driven Architecture, the term “metamodel” needs to be understood first. Definition according to [11]:

Metamodel – “a model that defines the structure, semantics, and constraints for a family of models.”

Model-Driven Architecture (MDA) [21] is a software design and development approach with models as main artefacts. It is built upon standards provided by Object Management Group. It put formal models and model transformations in centre of development process. Models are designed to support architectural separation of concerns. By doing so MDA aims to achieve portability, interoperability and reusability. Model-Driven Architecture is generally based on two things:

- a four-layer metamodeling architecture
- set of complementary OMG standards:
  - Meta-Object Facility
  - Unified Modeling Language
4.2 The Four-Layer Architecture

4.2.1 Generic Four-Layer Architecture

On Figure 4.1 a general four-layer modeling architecture is presented. It was inspired by MDA’s abstraction-levels concept. The bottom layer – M0 represents objects from the real world [3]. This layer includes all the things, that are represented by the models in the higher, M1 level. Such real-world’s objects representations can be more or less simplified and abstract, depending on model’s detaility. How expressive a model can be, is determined by corresponding metamodel from layer M2. The more concepts a metamodel provides, the preciser its models can be. Metamodels also have to be described in usage of some more abstract concepts. Those sets of concepts are provided by meta-metamodels from the most abstract layer of this architecture, M3. Meta-metamodel is simply a metamodel for defining other metamodels, including itself [11].

A modeling space is defined as: “modeling architecture defined by a particular meta-metamodel.” [11]. According to this definition the generic four-layer architecture is a foundation not only for modeling architectures based on object-oriented meta-metamodels, but for many various modeling spaces, for example RDF or EBNF [11].

Figure 4.1: Generic four-layer architecture (This figure is based on [11])
4.2.2 MDA four-layer MOF-based architecture

The MDA four-layer architecture is an instance of the generic architecture model, described in section 4.2. It comprises models and metamodels that are based on object-oriented MOF meta-metamodel.

The highest level in this architecture, M3 layer – meta-metamodel (MOF in this case) specifies an abstract language and framework which allows for defining, constructing and managing metamodels. This is a basis for both new-defined modeling languages (e.g. UML) and MOF itself, consistently with generic four-layer architecture. Metamodels, which were defined in MOF are placed in lower, M2 layer. Despite standard metamodels like UML, Common Warehouse Metamodel (CWM) or Software Process Engineering Metamodel (SPEM), it may contain also custom ones. At M1 layer models corresponding to metamodels from layer M2 and represent objects from reality, are positioned. And finally at the bottom layer – M0 are placed things from the real world. The difference in interpretation of objects in M0 layer is as follows: MDA assumes that real-world things are instances of models at M1, while newer approaches treat those models like “snapshot” of the real world. [3]

The four-layer architecture and corresponding standards are depicted on 4.2:

Figure 4.2: MDA four-layer MOF-based architecture (This figure is based on [11])

MDA approach and standards allows for [27]:

- specifying a system independently of the platform that supports it
- specifying platforms
- choosing a particular platform for the system
transforming the system specification into one for a particular platform

Those standards include:

- formalised way for defining models
- predefined models with described semantics and notation
- methods of transformation between models
- method for transformation from models to textual artifacts
- models serialisation and storage

According to MDA Guide [27] there are three primary goals of MDA: portability, interoperability and reusability realised through architectural separation of concerns.

MDA defines three different levels of abstraction on which systems can be analysed. Those levels can be equated to different points of view. At each level of abstraction a viewpoint model – system representation is defined [11]. Moreover MDA development process in which abstraction levels of a system are distinguished, let for separation of the specification of system from the implementation details, by usage of three main types of models which have different level of abstraction and platform independence:

- Computation Independent Model (CIM)
- Platform Independent Model (PIM)
- Platform Specific Model (PSM)

![MDA Workflow Diagram](image)

Figure 4.3: MDA workflow

Platform independence is a relative attribute of a model. Model can use features of selected platform, or may depend on some general concepts, that are available in most platforms, but some level of dependence always exist. Basic workflow
in MDA assume creation of CIM, PIM based on CIM, and transforming PIM to PSM. There are several approaches for that transformation, which can automate it in different degree. Today additional step is needed where PSM is transformed to code, in future, such step should not be necessary. Figure 4.3 represents the MDA workflow. On the basis of one abstract model – PIM, many different PSMs can be created. Then, from each PSM, the code in a specific languages can be generated.

4.3 Types of models

4.3.1 Computation Independent Model

Computation independent model is the most abstract model. The role of CIM is to describe environment in which system would be used and requirement that should be fulfilled. It also provides dictionary for terms that will be used in rest of the process. It should be understandable for domain experts, who have no knowledge about realisation of the system. CIM does not contain information about internal structure or other technical details of the system. Computation independent model can be provided in natural language and can be called domain or business model.

4.3.2 Platform Independent Model

Platform independent model is a precise description of a system structure and behaviour but does not include details concerning usage of underlying platform(s). Therefore it contains all the details that are not changing between different platforms. PIM is derived from associated CIM. This step is usually done manually. PIM can be modelled with general purpose modeling language or domain specific model, though UML is the most commonly used language. PIM should be described in MOF compatible model, to simplify model storage and transforming PIM to PSM.

In this thesis context PIM is a model (or group of models) that contains knowledge important in modernization process and should be transferred to new platform. PIM cannot use concepts that are specific just for one (“new” or “old”) platform.

4.3.3 Platform Specific Model

Platform specific model takes into account the platform specific issues of created system. Same system is modelled both in PIM and PSM, although models can have differences in structure. PIM describes system attributes in abstract fashion,
realised by one or many features on chosen platform what is modelled in PSM. Thus PSM shows greater complexity than PIM. If PSM provides all details needed to construct and run a system it is implementation. Otherwise it can be used as a PIM to construct more specific model.

In this thesis context PSM is a model (or a group of models) that precisely represents selected platform knowledge. It is strictly related with a concrete platform and represents its artifacts, so that the code for that platform could be generated directly from such a PSM without additional work or input.

4.4 The MDA technology base

To put model-driven concepts into practice OMG adopted number of existing standards and technologies. Namely [27], [37]:

- MOF
- UML
- XMI

4.4.1 Meta-Object Facility

The Meta-Object Facility (MOF) is a standard by OMG, which defines an abstract language that can be used to define interoperable metamodels and corresponding models. MOF technology is a foundation of the MDA environment [35]. It is object-oriented in nature, so it defines a syntax and essential elements for metamodels from which object-oriented systems can be constructed. MOF semantics define repository of metadata and provide services which support constructing, discovering and updating models. Moreover, those models, as long as they are defined in MOF metamodel, can be exchanged between application, transferred via network and serialised to XMI, a special XML dialect [28]. Many OMG standards refer to “four-layered metamodel architecture” concept, nevertheless number of the layers may be different. In fact MOF allows for any number of layers, but at least two – to be able to navigate from object to its instance. Example of an architecture consisting of two layers are generic reflective systems, where Object is an instance of Class (Class is on a higher level of abstraction). Relational database systems are based on three-layered architecture SysTable/Table/Row and UML 2.0 Infrastructure is based on the most typical – four-layered architecture: MOF/UML/User Model/User Object [29].

The MOF specification includes the following [37]:
• an abstract model that describes MOF itself. It contains essential set of objects and associations between them
• mapping rules for transformation of any MOF-compliant metamodel to language-independent interfaces. An implementation of these interfaces allows for modifying models derived from that specific metamodel.
• Reflective interfaces for those of MOF-based metamodels for which the mapping interfaces are unknown. Those reflective interfaces enable to modify and discover the metamodels anyway.
• Rules which define for elements from MOF-based metamodels, semantics of:
  – life cycle
  – composition (i.e. inheritance, nesting)
  – closure

4.4.2 Unified Modeling Language

The Unified Modeling Language [15] is a most commonly used MDA metamodel. It is a general-purpose modeling language for software engineering, though parts can be useful in different domains. It consist of abstract syntax and visual concrete syntax. UML is based on MOF, therefore it have potential of interoperability. In practice, today’s tool experience many issues during interchange of UML models, especially diagram interchange does not occur. Object constraint language (OCL) [31] is important part of UML standard. It is a declarative formal language that allows navigation and definition of constrains on model or metamodel level that can not be declared in metamodel semantics. Correct model must meet such rules. Although OCL was created specially for UML, it has been refined, thus one can use OCL with any MOF-based model. OCL can additionally provide semantics for behavioural aspects of model.

4.4.3 XMI

XML Metadata Interchange is an OMG standard for mapping the MOF to the XML by W3C. XMI defines how to represent MOF-based models with usage of XML tags. Metamodels are translated to XML Document Type Definitions (DTDs), while models are translated directly to XML Documents, consistent with corresponding DTDs. Thanks to XMI, MOF-compliant models and metamodels can be serialised in a formal way. It solves many problems connected with usage of tag-based language for describing objects and their associations. What is more,
both serialised metadata (in form of tags) and instances (tags content) can be stored together in the same document, because they are all described in XML. It improves readability, because concrete objects can be understand through their metadata.

## 4.5 Transformations in MDA

Transformations are a key concept in MDA. CIM has informal nature, thus transformation from CIM to PIM is done manually. Code generation is not a new concept, it is straightforward and supported by many tools. The key role of model-to-model transformations in MDA workflow, distinguishing MDA from other approaches. Typically first the PIM is built, then one or many platforms are chosen because of business concern, or to provide the best implementation for a PIM architecture. When the platform is chosen, transformation which maps PIM artifact to platform features is made. This step needs some decisions about the design of a PSM model, some of then can be marked on PIM (models provide a special way of doing this, e.g. the UML provides stereotypes), other are described in transformation languages. Transformation languages must be a formal language to allow its interpretation by transformation tools. OMG defines two standards for definition of M2T and M2M transformations.

### 4.5.1 MOF Model to Text Transformation Language

MOF Model to Text Transformation Language (MOF2Text) is an OMG standard which defines template language for code generation. It uses subset of OCL [31] to query MOF models. It reuses parts of QVT specification to support definition of side effects free functions that simplify code generation. MOF2Text support traceability.

### 4.5.2 MOF Query/View/Transformation

MOF Query/View/Transformation (QVT) standard defines languages for validating and transforming any number of EMOF [29] compliant models in any direction. Transformation conforming to QVT can have PIM as an input and PSM as an output model. QVT languages correspond to the EMOF metamodel. It is even possible to create transformation, which constructs a transformation model and executes it. QVT standard defines three languages:

- QVT Relations (declarative)
- QVT Core (declarative)
• QVT Operational Mappings (imperative)

They are used to describe model-to-model transformation in MDA context. Each consist of concrete textual syntax and a metamodel with well-defined semantic. Additionally two graphical representations have been provided for the relations language. To support complex transformations, QVT allows for using black box operations (described below) and a hybrid approach – using relations together with operational mappings. In such cases transformation is unidirectional.

QVT Relations

QVT Relations is a declarative, bidirectional language with automatic support for tracing. It can work on any number of models. In the relations language, a transformation between candidate models is defined as: “a set of relations that must hold for the transformation to be successful” [34].

QVT Core language

Core language is equal in semantics to QVT Relations, but it is on lower level of abstraction. It needs more effort to create same transformation and it is not intended to be used directly. QVT standard defines detailed semantic of QVT Relation with full mapping from Relational to Core language in Core language. This mapping can be used as QVT Relations compilation step.

QVT Operational Mappings

Operational Mappings language extend OCL with imperative constructs, thus it is an imperative language. However it reuse elements of QVT Relations. Operational mapping is semantically a refinement of relation that share some elements with the mapping, but has additional imperative semantic. Tracing support is enabled, but language is unidirectional.

Black box operations

Black box operations define a way to plug-in any implementation of given MOF Operation signature. It allows to implement complicated algorithms, using any language with defined MOF bindings. It allows also to use domain specific libraries. For instance domains like mathematics or bio-science, in which it would be difficult or even impossible to express some domain-specific algorithms using OCL. If black box operation must work on model elements, then API for model manipulation is needed. On the other hand, black box operation may simply return a primitive value(String, Integer), or provide additional trace options. All black box operations
are treated as imperative, hence their usage in Relations language forces hybrid mode.

### 4.6 Architecture Driven Modernization Initiative

OMG is working on standard called Architecture-Driven Modernization (ADM), which systematises the process of modernization with usage of standards created for MDA. The OMG in [13] defines ADM as a “process of understanding and evolving existing software assets for the purpose of

- Software improvement
- Modifications
- Interoperability
- Refactoring
- Restructuring
- Reuse
- Porting
- Migration
- Translation into another language
- Enterprise application integration
- Service-oriented architecture
- MDA migration”

#### 4.6.1 ADM goals

There are several purposes of why ADM was bring into existence. The main, ultimate aim of ADM initiative was revitalising existing applications, which are difficult to maintain and evolve. The second reason is a need to make the software more agile and adaptive. Another goal is to expand and influence other OMG standards in modeling, especially those, which are part of the MDA initiative and the MDA itself. What is more, the next goal of ADM is to consolidate best modernization practices, to make the process less risky.

Hence, deciding to use ADM for modernization, companies gain improved productivity of software development and bigger return of investment with smaller effort and lower cost.
4.6.2 ADM process

Modernization project scenarios can take numerous forms, but in general they can be classified as one of three perspectives, depending on the architecture domain which the project influences [23]. These perspectives are: business architecture, application/data architecture and technical architecture. The first one is classified in a range of business domain, and the last two in IT domain. The business domain is represented by models, diagrams of business processes and rules, hence IT architectural domain represents technical architectures, traditional data and application. The left side of Figure 4.4 represents existing solution (as-is state) and the right side, the target solution (to-be state). Architecture transformations can be done on different levels of abstraction, depending on the modernization scope. It can also be done incrementally. Architecture transformations can be classified as:

- Business Architecture-Driven Modernization
- Application/Data Architecture-Driven Modernization
- Technical Architecture-Driven Modernization

This classification is also shown on Figure 4.4. These modernization types are respectively signed as symbols: “B”, “A” and “T” on this picture, and on Figure 4.5, Figure 4.6, Figure 4.7.

Anyway, regardless of the level of modernization impact, the following three elements are always invariable [23]:

- Discovering knowledge from the existing solution, required to understand the application artifacts
• Defining target architecture – it is necessary to define a transformation approach. Created target solution serves as a framework, the existing one is mapped into.

• Transformations from existing into target state. These mappings can range from physical level, like migrations between languages, to the more abstract ones, e.g. transformation from business rule into an environment based on rules.

Technical Architecture-Driven Modernization

![Figure 4.5: Technical Architecture-Driven Modernization](image)

Technical Architecture-Driven Modernization is a very popular type of work among companies. It may be for example a migration of software system from one platform to another or transformation between two languages. Technical architecture modernization projects should be distinguished from Application/Data ones; in the second option, enhancements impact in on system-level or data design [23]. For example when a legacy system, written in a procedural language is mapped to an object-oriented language – in such a case, changes in infrastructure imply changes in application design. It does not involve any other layer than the technical one, what is depicted on Figure 4.5.

Application/Data Architecture-Driven Modernization

Application/Data Architecture-Driven Modernization involves bigger scale projects than the previous one. They require changes in technical and application/Data layer [Figure 4.6] In this category most of projects with redesigning or redeploying
a MDA-based system, are clustered. Motivations to make this kind of modernization are many. Very often such a need emerges when the old solution can no longer be up to the growing business needs through performing incremental maintenance activities. Another example was mentioned while describing previous approach, namely, migration between different platforms, what forces to reorganise the modernized application architecture. Such projects incorporate both data, application and technical layer. Alternatively, along with new emerging business needs that will require making changes in the application model, a need for reorganisation the data structure may also be forced [23].

**Business Architecture-Driven Modernization**

Business Architecture-Driven Modernization projects are the most complicated
and comprehensive ones. They incorporates models on all levels of abstractions – business, application and data architecture, and technical architecture models. Anyway very little modernization projects of this kind have been completed successfully, because of the fact, that on such high level of abstraction there is a lack of standardization in model transformations between business and IT domain.

### 4.6.3 ADM Standards

Summing up, as it is shown on Figure 4.8, ADM process consists of two phases – in the first one reverse engineering techniques are used and in the second one – forward engineering. In the preliminary phase requirements toward the target application are gathered, in order to choose the most suitable modernization strategy: technical, application/data or business architecture-driven (Figure 4.8 presents the business architecture-driven approach). From the source application knowledge is “mined” and represented in a form of model. Thus, to perform this step a kind of model discoverer is needed. The next step is raising an abstraction to a necessary level, if needed. The highest level of abstraction is required, when changes in business rules are planned. When the required abstraction level of the model is reached, it is upgraded into the target model. Because raising the level of abstraction and upgrade requires transforming a model, to perform these two steps a tool for model manipulation is necessary. When the appropriate target model is all ready, the forward engineering part begins. The abstract model is specified and platform-specific artifacts are added. At the end, the target application code is generated. Figure 4.8 depicts only a general process of modernization according to ADM assumptions, the upgrade phase may also be performed on an application model level, if no changes in business domain are provided for.
In a range of ADM project several standards were created. The most important is Knowledge Discovery Metamodel (KDM) and Abstract Syntax Tree Metamodel (ASTM) – they are described in subsection 4.6.5.

### 4.6.4 Knowledge Discovery Metamodel

Knowledge Discovery Metamodel is a metamodel, that represents information about assets of existing software and their operational environment. It serves for exchanging application metadata by modernization tools, provided by different vendors, across different languages, applications, platforms.

The KDM provides features not only to represent code, but entire software systems. It uses entities with attributes and relations between them to model both structure, code organisation, data tables, data tables content and behaviour of an exiting system. Nevertheless the KDM does not represent the system below a procedural level (it can represent assignments, but it can not represent loops or statement order). For this purpose Abstract Syntax Tree Metamodel was created.

The concept of containers is also important. Containers are entities that can include other entities or containers. Thanks to this idea the KDM models can represent a system on different granularity degrees.

### 4.6.5 Abstract Syntax Tree Metamodel

The aim of Abstract Syntax Tree Metamodel (ASTM) is to provide universal representation of programming languages, that can be used in ADM (especially in metrics and modernization scenario). ASTM allow to model existing system below the procedural level. Thus, it can complement KDM to allows detailed representation of entire software system. Moreover, it does not depend on the KDM and can be used alone. ASTM can be also used as PSM and KDM can be derived from it as higher level view of the system (PIM). ASTM standard defines Generic ASTM (language neutral – corresponds to PIM) and Specific ASTM (that use metadata and is specific to one language – corresponds to PSM). At the time of writing ASTM have not been finished.
Chapter 5

Model-Driven Software Modernization

Author: Krzysztof Kowalczyk

Model-Driven Software Modernization is a term proposed for the purpose of this master thesis. It describes a group of approaches toward modernization of legacy code. In this chapter we will present two identified MDSM approaches: The “reverse” MDA approach (RMA), which actually exists and occurs in some commercial modernization projects, but it has not been systematized yet, and the second one: The Direct Transformation Approach (DTA), which was proposed by us, because we suppose that it may be better than the RMA, for some specific modernization projects. In the section 5.5 the characteristics of projects, that each of the MDSM approaches is suitable for, are detailed. Section 5.4 contains the description of tools which are needed to perform each step of the MDSM process.

5.1 Model-Driven Software Modernization in general

In this thesis MDSM is defined as a process of a partially automated modernization of legacy code, that is based on models and model transformations. Its main aim is to lower an effort of a modernization, especially while modernizing many similar software products. The MDSM uses tools and processes that are related to MDE, MDA and ADM in particular:

- Metamodeling concept – to define and manage models
- Text-to-model transformation – to read existing code
• Model-to-model transformation – to realise a modernization logic
• Model-to-text transformation – to generate code of a new application
• Workflow engine – to define modernization steps

5.1.1 MDSM assumptions

In this section, our MDSM assumptions are gathered, to present a general characteristic of an MDSM-based approach. Such an approach should apply the following principles:

1. Reusability – If possible, modernization itself and infrastructure components should be reusable.

2. High level of automation – Most of the tasks should be automated. This supports reusability of transformations and may allow for iterative development.

3. Raising quality of code – It means that new, modernized code characteristics should be improved.

4. Keeping existing functionality – Important knowledge gathered in code should not be lost. Direct changes to functionality during modernization is possible, but it is not in the scope of MDSM. MDSM can introduce indirect changes to functionality if new platform, library or architecture as a side effect (web access, new persistence options etc.). Those changes should not change the semantic of the application.

5.1.2 Model-Driven Software Modernization Process

We based the process of the Model-Driven Software Modernization on the migration plan introduced in ISO/IEC 14764:2006 standard [21]. According to [21] the migration process contains the following steps:

a) “Requirements analysis and definition of migration;
b) Development of migration tools;
c) Conversion of software product and data;
d) Migration execution;
e) Migration verification;
f) Support for the old environment in the future.” [21]
The MDSM assumes that the old system will be replaced with a modernized version, so there is no need for support for the old one, that is why step “Support for the old environment in the future” was omitted. Very important, from MDSM’s point of view is step 2: “Development of migration tools”, due to the fact, that there are not any ready-made tools designed for this concept, so choosing the most suitable and compatible with each other tools and standards is crucial to modernization efficiency. The substep “Handmade correction” is provided for those parts of modernization, which would be unprofitable to automate. It could be for example a part of code, that is very specific for a modernized application – in such case the complicated transformation, written to map this part, probably will not be reused. Of course, this substep is optional.

The MDSM Process:

Step 1 Requirements analysis;
Step 2 Development of migration tools:
   a) Identification of needed migration tools:
      i. A metamodel standard
      ii. A model discoverer
      iii. A model manipulator
      iv. A code generator
   b) Acquiring and integration of migration tools;
Step 3 Target platform preparation;
Step 4 Definition of an automated migration;
Step 5 Migration execution:
   a) Transformation execution;
   b) (Optional) Handmade correction;
Step 6 Migration verification;

5.2 The “Reverse” MDA approach (RMA)

This section includes a description of the first identified MDSM approach, based on the idea of “reversing” the MDA process, described in chapter 4. In particular we present the workflow of RMA modernization, two different variants of this approach and motivations for using this type of modernization.
5.2.1 The RMA workflow

As was mentioned before the RMA process is based on the MDA workflow, nevertheless part of this workflow is reversed – in the initial step, a model corresponding to the legacy platform metamodel (PSM1) is obtained from the legacy code, as shown on Figure 5.1. In the second step this model is transformed into a PIM - an abstract model, independent from both the old and the new platform. While having a PIM, the regular MDA workflow can be performed, so using M2M transformation, a PIM will be mapped into a PSM2 and then the PSM2 will be transformed into code (an M2T transformation). The MDA vision \[24\] assumes that there are existing transformations that can be reused to map a PIM to a PSM. Thus, it should be possible to generate models for many different platforms as in the regular MDA workflow. In theory, such an approach would be well suited for software modernization as long as we can obtain a PIM from existing application.

PIM keeps information about the application without any constructions specific to one platform. But the meaning of “platform” and “platform independence” depends on a situation. A platform can be a different version of the same language, library or different but similar languages (e.g. two object-oriented languages). It can also be a language or a library that represents different programming paradigms (e.g. object-oriented, aspect-oriented, procedural).

Summing up, the general RMA workflow, which is depicted on Figure 5.1 consists of four stages:

1. Model discovery – transformation: code → PSM1
2. Raising the model’s abstraction level – transformation: PSM1 → PIM
3. Specifying the abstract model – transformation: PIM → PSM2
4. Code generation – transformation: PSM2 $\rightarrow$ code

5.2.2 RMA with Essential PIM (RMA-EPIM approach)

Essential PIM (EPIM) represents elements that have been identified as the most valuable and which are only a part of the application. For example, EPIM would represent business rules, persistent entities or a domain model that has been reengineered from the application code. Metamodels like BPMN or less likely UML can be used as EPIM and focus on a specific fragment of the application. In ADM, Business Architecture-Driven Modernization, as described in [23], represents an approach similar to RMA-EPIM. There is small difference though, it seems (work is not yet finished and not all specifications are public) that ADM assumes that some handmade changes to the business logic of application can be done in Business ADM (by manipulating a model on the highest level), those changes may be the purpose of modernization. Such handmade changes are possible in MDSM, but they are we do not specify them in MDSM process. MDSM purpose is to migrate an application to a different platform (different technology, new version of language, library etc.) or to change the structure of the application to increase chosen code characteristics (e.g. interoperability).

Distinction between two extreme occurrences of this approach has been made:

a) RMA with Complete PIM (RMA-CPIM approach) Usage of a Complete PIM model is another approach toward MDSM. The Complete PIM (CPIM) is a PIM, that models an application on a lower level of abstraction than EPIM. In practice it means that more details from the application code are reflected in the model. Complete PIM can be represented by an UML model (much more detailed one, than to represent EPIM), possibly including UML profiles. KDM together with ASTM [33] or ASTM itself can be an alternative to UML models. They allow for precise description of the semantic of methods. Dedicated MOF metamodel can be also used as a CPIM. Sodifrance corporation uses as a PIM their own metamodel called ANT, it was described in [12]. ANT may also be considered a CPIM, because it allows for modeling data structures, imperative behavior, navigation in application and user interfaces. Moreover, Application / Data Architecture-Driven Modernization concept [23] is very similar to the CPIM approach.

5.2.3 Summary of the RMA

In section we consider what kind of tools are needed to perform each of these steps.
Motivation

Motivations toward this approach:

- PIM can represent only a part of a legacy system, which is claimed to be valuable by stakeholders (e.g., business rules, domain model).

- Usage of PIM simplifies migration to many platforms, according to MDA assumptions (the ability to change target platform).

- Existing PIM to PSM and PSM to code transformations from MDA can be reused.

- Similar approach is under the standardisation process by the OMG initiative – Architecture Driven Modernization.

- Such an approach has been successfully applied to a real industrial modernization project [12].

5.3 The Direct Transformation Approach (DTA)

In the second solution a PIM model is not created, instead the direct transformation between two platform specific models is used. As it is shown on Figure 5.2 the first step is the same as in the previous approach, so with usage of a text-to-model (T2M) transformation, the PSM is obtained from the legacy code. The second step is different – instead of transforming the PSM1 into a platform independent model, it is directly transformed into the PSM2 – a model specific to the target platform. The last step is also the same as in the first approach - using
M2T transformation the target platform code is generated. In this approach, the abstraction level of models do not change unless platforms abstraction levels are different.

This approach toward modernization is non standard. Anyway, small transformations, or transformations that are not based on MDE use similar approach. We have decided to consider this approach, because it seems as better solution for some kind of modernization projects (e.g. projects on the Technical level in ADM and partially on the Application/Data level [23]) when availability of existing MDA components is very limited.

5.3.1 Summary of the DTA

The DTA workflow

The general DTA workflow, which is depicted on Figure 5.2, consists of three steps:

1. Model discovery – transformation: code → PSM1
2. PSMs transformation – transformation: PSM1 → PSM2
3. Code generation – transformation: PSM2 → code

Motivations

Motivations:

• We have said before, that MDA assumes existence of transformation cartridges – reusable components which include the definition of specific transformations from PIM to PSM (e.g. from PIM represented by a UML model, to a model specific to the java, or .NET platform), or from PSM to code (e.g. from a java model to java code). Anyway, in practice it occurs that appropriate transformations do not exist. Thus, it seems reasonable, that developing one transformation (PSM1 → PSM2) is easier and less work-intensive than developing two transformations (PSM1 → PIM → PSM2). It is caused by the fact, that each entity needs to be handled twice in the RMA, whereas in the direct transformation they need to be handled only once. Furthermore, when source and target metamodels have common parts, some of them may even do not need to be changed.

• There is no need for PIM. In the RMA a problem can occur in acquiring appropriate metamodel for PIM, in the DTA we will not have no such problem.
5.4 Common tool demands on the identified approaches

Identified approaches have similar processes and a number of common requirements. One of the substeps of the second step of the MDSM process, which is: “Identification of needed migration tools” (Section 5.1.2), partly overlaps, when considering these two approaches. As it is depicted on Figure 5.3, the common processes are: the model discovery and the code generation, so tools that support them are necessary. Mappings between models are different, because the RMA requires transformations from PSM1 to PIM and from PIM to PSM2, while the DTA only a transformation from PSM1 to PSM2, nevertheless these are M2M transformations, so a tool providing them is also required in both cases. Additionally, a metamodel for PSMs has to be chosen in both cases. Summing up, the both approaches need migration tools for model discovery, model manipulation and code generation, and a metamodeling standard. The Table 5.1 presents more specific requirements towards migration tools, typical for each of the identified approaches.

![Figure 5.3: Common parts of the RMA and the DTA workflows](image)

5.4.1 Metamodel

We consider a metamodeling standard to be the most important foundation of the MDSM. Each step of its process can be realised with some general purpose solution or dedicated language/technology. Models can be represented by classes in object-oriented general purpose language. Nevertheless a standard solution for describing models and metamodels, with ability to store them and dedicated API for manipulating them, allow to reuse all available standard-compliant solutions.
### Identification of needed migration tools

<table>
<thead>
<tr>
<th>MDSM approach</th>
<th>DTA</th>
<th>RMA</th>
</tr>
</thead>
</table>
| Metamodel standard | 1. For PSM1  
2. For PSM2 | 3. For PSM1  
4. For PSM2  
5. For PIM |
| Model discoverer | 1. code → PSM1 | | |
| Model manipulator | 1. PSM1 → PSM2 | 1. PSM1 → PIM  
2. PIM → PSM2 |
| Code generator | 1. PSM2 → code | | |

Table 5.1: Identification of needed migration tools for the RMA and the DTA

in the process. MOF [29] and XMI [28] are addressing this demand. The first approach – the RMA, requires more metamodels than the DTA, because of one extra model, which is required in this technique – a PIM. Thus, as it is shown on Table 5.1 the DTA requires only two metamodels – one for a PSM1 and the second for a PSM2, while The RMA needs three of them – for a PSM1, a PSM2 and a PIM. They do not have to be different metamodel, PSMs and PIM may be simply different models of the same metamodel.

#### 5.4.2 Model discoverer

First step in the MDSM is similar in each solution. The model of the existing application need to be created based on existing artifacts. Model discovery is a specific form of reverse engineering, where the source code of application is transformed into a model specific to source platform. Hence a platform specific metamodel of examined platform is needed. Artifacts like database schema or XML files are relatively easy to process as they have dedicated API to read and manipulate its content. This is not the case with application code. Hence model discovery is connected with a problem of code parsing.

Platform specific metamodels can represent different levels of abstraction. Generic metamodels like KDM [32] and ASTM [33] can be used, but in fact, the more generic a metamodel is, the more complex models corresponding to this meta-
model are. So models based on a more platform-specific metamodel should be less compound. Because of the fact, that this step is common for all MDSM approaches, there is no distinction between the RMA and DTA – both of them need a tool for transforming code into a PSM1, what is showed in Table 5.1.

### 5.4.3 Model manipulator

Once the models are available, they need to be manipulated. This can be done with General Purpose Language (GPL), DSL, mapping models or combination of such tools. Each modernization approach need at least one tool for model manipulation. GPL can be used, as usually appropriate API exists, but usage of a language dedicated for M2M transformations, can greatly improve productivity in development of transformation. The QVT [34] standard addresses the need of this kind of transformations. The number of needed kinds of model manipulation tools is dependent on types of metamodels used to represent PSMs or a PIM. As it is shown in Table 5.1, generally the DTA needs only one type of transformation: from a PSM1 to a PSM2, while the RMA two of them: from a PSM1 to a PIM and from a PIM to a PSM2, unless PSMs and a PIM are represented by the same metamodel.

### 5.4.4 Code generator

In the last step code is generated from Platform Specific Model. Last step is similar in each MDSM approach and MDA vision, thus code generator cartridge should exists (as well as the Platform Specific Metamodel). This implies a need for a tool that supports generation of code from a model. It can be realised by DSL, a general purpose template engine or visitor pattern [3.2.2]. MOF2Text is a standard of a model-to-text transformation language, that addresses a need of code generation. Similarly like the model discovering, this step is common for both approaches, so both of them need a one tool, which would generate code from a PSM2. This is also mentioned in Table 5.1.

### 5.5 Suitability of MDSM approaches

#### 5.5.1 RMA with EPIM

EPIM model can not be used if technical details about application are needed. But it is good approach when only application logic, domain model, relations etc. are interesting for stakeholders. For example, application is migrated to new platform and there is a need for redesign of user interface, changes in data persistence,
change of language etc. and existing application has valuable domain model (e.g. database) that can be extracted and reused. EPIM is connected with high amount of handmade code, or with sophisticated MDA process.

5.5.2 RMA with CPIM

CPIM is good when more detailed model of application is needed and there are many target platforms, some available transformations, or new application will be developed in MDA way. We claim, that DTA is better when there is no proper PIM/PSM transformations available.

5.5.3 DTA

DTA approach is most effective when there is no need to use different metamodels for PSMs in modernization process. Then it can take less than half of RMA with CPIM effort. If there are different PSM metamodels and no appropriate MDA infrastructure, it should be take about half of RMA with CPIM approach, as in RMA same metamodels elements need to be handled two times, what double the effort.
Chapter 6

Feasibility study

In this section an exemplary modernization problem will be introduced and solved by applying the MSDM techniques – the DTA and the RMA. Additionally, general problems that need to be solved before implementing modernization will be discussed. This requires analysis of existing tools for discovering models, model manipulation and code generation, and possibilities of their integration, because there is not any complete tool, which would provide all the necessary functionalities. Knowledge of technologies have been gathered by literature survey and practical experiments with the tools. Eventually a representative part of modernization will be implemented to prove its feasibility.

6.1 Introduction

Author: Anna Kwiecińska

Model-Driven Software Modernization can be seen as a process of source code translation between two different platforms. Nevertheless, according to subsection 2.3.4 modernization is not always connected with a change of a platform. One of the modernization problems that we identified in industry is migration from an older to a newer version of the same framework. In such case, only a part of code need to be changed, and this type of modernization projects we will consider in this thesis.

6.2 Description of the exemplary modernization project

Author: Anna Kwiecińska
This exemplary project is a proof of concept which demonstrates feasibility of the MDSM concept in general, and the MDSM’s DTA and the RMA in particular. It is a practical application of these approaches, conducted according with the MDSM’s process steps, introduced in subsection 5.1.2.

The aim of this task is to conduct modernization of an application written in Java 1.4 and Hibernate 2.x framework, which uses XML mappings, into Java 1.5 and Hibernate 3.x which uses Java Persistence API (JPA) annotations. This is depicted on Figure 6.1. Modernization should be automated and no functionality should be lost.

The source code, used in the example is taken from CavetEmptor – the application for online auctions, that is available on the LGPL License [16]. This is an exemplary project created for the book “Hibernate in Action” [4]. It is written in Hibernate 2.1.6 and uses many of its features, even those which are not typically in use in real projects, therefore it is a good base to migrate, because it emulates the complexity of real industrial problems, quite well. Moreover, usage of both standard and non standard Hibernate features raises the level of migration reusability, so the next modernization project will not require many changes in existing transformation code.

This choice was motivated by the fact, that this is a frequent problem in industry. There is a lot of software, that uses Hibernate 2.x, but at the moment Hibernate 3.3 is a standard, so it would be better to modernize it to Hibernate 3.3.

Figure 6.1: Main task of planned modernization.
6.2.1 Description of the modernization context

*Author: Anna Kwiecińska*

Hibernate is an open source Java framework used extensively in industry. It is designed for persisting business models defined as Java classes in relational databases with usage of object-relational mapping (ORM). Hibernate 2.x uses Java 1.4 language and mappings to relational database defined in XML files. Growing popularity of Hibernate and other ORM frameworks in industry caused the need for standardisation of such approaches, thus Java Persistence API (JPA) - an object-relational mapping standard, was created.

Hibernate 3.x is a JPA implementation, moreover it uses new language features from Java 1.5, e.g. generics, which increase code quality and reduce code complexity. JPA standard uses annotations that allow for defining mapping as type safe metadata and can replace XML mapping files required in Hibernate 2.x. Annotations are the most important difference between Hibernate 2.x and 3.x. Furthermore, new version of Hibernate is binary incompatible with older. They use different namespaces Java packages) and there are some changes in API.

![Figure 6.2: Relations between Java 1.4 and Java 1.5.](image)

The selected modernization problem is distinguished by similarity of a source and target platform. Java 1.4 is semantically a subset of Java 1.5 as shown on Figure 6.2. Java 1.4 code is a valid Java 1.5 code with small exceptions so the same metamodel can be used for both platforms and most elements remain unchanged. This has potential to simplify the migration. Most elements from the source code can be copied without change to target code. Additionally, small changes in the target model need to be done as the structure of Hibernate libraries have changed, and that need to be reflected in code. Moreover, transformations that do not alter behavior of a code, but increase code quality are planned.
6.2.2 Classification of the problem

Author: Anna Kwiecińska

This problem represents a specific branch of legacy system problems. Namely those, which use the old version of a framework, but need to be compatible with a newer version of the same framework. This modernization problem appears quite often in industry, usually due to lack of compatibility – emerging technologies, which provides new helpful features, are compatible only with the newest versions of existing frameworks or libraries. Therefore, demand for this kind of modernization is big. Moreover, cost of such a transformation can be relatively small in comparison with, for example, migration between two different language platforms, because the process might be automated to a large extent. Figure 6.3 shows the scope of this research using Seacord et al. taxonomy of system evolution. In this nomenclature, it was classified as modernization, more specifically – one type of reengineering called source code translation. This category consists both of migration between two languages and between two versions of the same framework or language. In this research only the second type – migration of code written in an old version, into newer version of existing framework is considered.

![Figure 6.3: Problem classification using system evolution taxonomy by Seacord, Plakosh and Lewis.](image)

ISO/IEC 14764 standard provides a different classification of modifications into software systems. In contrast to the classification from the Seacord’s et al. book, this division is based on reasons of changes and expected business value after the change is made, rather than the scope of changes. According to this division, area can be classified as preventive as well as adaptive and perfective maintenance. Preventive, when it is planned to expand in the future the existing system and make use of some new features, which are not compatible with old versions of framework, that is still in use by the system.
6.3 Exemplary modernization project: the RMA

Author: Anna Kwiecińska

Step 1: Requirements analysis

The migration requirements specific for this project:

 Req 1 XML mapping files from Hibernate 2.1.6 shall be replaced with annotations from Hibernate 3.3.

 Req 2 Hibernate 2.1.6 namespaces (“net.sf.hibernate.*”) shall be replaced with Hibernate 3.3 namespaces (“org.hibernate.*”).

 Req 3 New methods, which were introduced in Hibernate 3.x, of existing interfaces, shall be implemented.

 Req 4 Occurrences of obsolete methods from Java 1.4 shall be replaced with recommended methods from Java 1.5.

According to MDSM principles the target, generated code should have higher quality than existing code of the legacy application. For that reason we decided to implement some refactorings, that do not affect functionality of the system, but raised code quality.

It is worth to state explicitly, that some common requirements are not necessary in given example:

1. **Transforming each element** – When two platforms are different, migration need to handle each element of a source model and map it as an element of a target model. When target and source models have a common part in this modernization, not all elements need to be changed. These which are common, can be copied or in-place transformation can be used.

2. **Generic transformation** – Although reusability of components is important, generalization of transformations is not needed. A transformation does not need to handle all combinations of each element from metamodel, but only those that occur in a source model. If one would like to apply the MDSM to a different application with the same problem (migration from Hibernate 2.x to Hibernate 3.x) some additional work may (and most probably will) be needed, but it would need potentially a lot less effort than implementing a generic transformation.
3. **Two different metamodels** – Java 1.5 is a superset of Java 1.4, so only one metamodel could be used to model both platforms, PSM1 and PSM2 are simply two instances of the same metamodel.

**Step 2: Development of migration tools**

According to the MDSM process, this step consists of the following substeps:

a) Identification of needed migration tools:

i  A metamodeling standard

There is a number of different metamodeling standards and techniques. The selection of a metamodeling should be conditioned on its completeness – so that it could handle to describe the whole legacy system on required level of abstraction, and on tools availability. These tools should be compliant with the selected metamodeling solution, so there should be a dedicated code generator and model transformation language. Usage of such tools would significantly lower the effort required by the transformation. Moreover, XML integration is a one step of this exemplary migration, so if the chosen metamodeling solution supports XML-related technologies, the modernization effort will be even lower. Also support for MOF and XMI standards would be a plus.

ii  A model discoverer

There is an explicit need for a detailed platform model, what implicates a detailed (thereby quite sophisticated) model discovery phase. Reusability of components is also needed. Eventually a Java 1.5 ASG-like metamodel would be recommended, so that one PSM can be used for both platforms. Additionally, it would be easy to discover, as a PSM1, and pretty straightforward to generate Java code on its basis – as a PSM2. A metamodel and a discoverer for Hibernate mappings has to be provided also, if XML cannot be adapted as a model.

iii  A model manipulator

According to the MDSM principles tracing and bidirectionality is not needed for this transformation, so in theory, most of available M2M technologies should be sufficient. Nevertheless, if metamodeling solutions do not support XML, then a M2M language should support it. In other case an extra integration effort, or a model discovery phase would be needed. Additionally, \[\text{LS}\] indicates that declarative languages do not handle in-place transformations well.
iv A code generator

It has been identified, that not all changes can be done automatically, therefore, support for handmade changes must be provided for in the modernization workflow. Most of M2M technologies provide some kind of customisation of M2T transformations, usually it takes form of protected code regions, where changes to the generated code can be added. But it would not be enough in this case, because changes may be necessary in any place in the code. A possibility to change any fragment of generated code and preserve such change between executions of transformation would be a plus.

b) Acquiring and integration of migration tools;

i A metamodeling standard – Eclipse Modeling Framework

ii A model discoverer – Xtend integration with Java and XML (Eclipse Xpand)

iii A model manipulator – Xtend (Eclipse Xpand)

iv A code generator – Xpand (Eclipse Xpand)

Justification of such nonstandard selection of technologies is provided in next chapter.

Step 3: Target platform preparation

The aim of this step was configuration of the target platform, to which the legacy application was supposed to be migrated. As it was said before, our target platform is Java 1.5 and Hibernate 3.3, while our source platform – Java 1.4 and Hibernate 2.1.6. In details, configuration needed a replacement of the following libraries:

1. commons-collections-2.1.1.jar → commons-collections-3.1.jar

2. dom4j-1.4.jar → dom4j-1.6.1.jar

3. ehcache-0.9.jar → ehcache-1.2.3.jar

4. hibernate-2.1.6.jar → hibernate3.jar (Hibernate 3.3.1 version)
Step 4 and 5: Definition of an automated migration; Migration execution

Introduction and problems with the RMA

This step should be consistent with a workflow, specific for each MDSM approach, so it is not common to all of them. Thus, in this case the migration process is based on the RMA workflow section 5.3. Specificity of this modernization project required introducing some changes to the workflow, they are described farther in this section, nevertheless its general concept was preserved.

Simplified model (EPIM) When we were starting this research, there was not any detailed model discoverer, thus the direct approach was considered to be too complex. We have chosen an EPIM-like model as a base for the transformation. Anyway, we were not able to model behaviour The model was enriched with not parsed data in a form of tags (mostly method body content has been put there). It is a compromise. By using EPIM-like model the T2M process is simplified and existing parser will be reused. UML2 (EMF implementation) has been chosen for PIM model, especially class diagrams related fragment. Without these tags all the information about a system behavior (method body, field initialisation etc.) would be lost.

Shortcomings Such an approach has significant shortcomings. As a huge part of the code will not be represented as a model, its semantics is not known and it is hard to change such code. A structure is modelled, thus it can be changed. In particular, an addition of annotations is possible, but other changes may corrupt model. Nevertheless the main task, the mapping from XML files to annotations is relatively easy.

An approach type The discussed prototype does not exactly represent the RMA with EPIM model. There is no target PSM (PSM2) and code is generated directly from PIM as shown on Figure 6.4. Abstraction level of model does not change much, but first transformation be considered as M2M. It also does not conform to the direct transformation approach. Such approach was possible only because code is added and not changed. This is exceptional in modernization. Eventually it have not been considered solution to given problem and have been abandoned.

A sketch of this prototype implementation is shown on Figure 6.4. Each of phases presented on this figure is described in the following sections.
CHAPTER 6. FEASIBILITY STUDY

Model discovery

In this step source model discovered from code and XML files. Source is represented by the metamodels of the legacy platform:

- Java structure metamodel (Eclipse JDT IJavaModel)
- Hibernate XML mappings metamodel

Thanks to selection of these metamodels, there is no necessity of the model discovery phase, as infrastructure for that is a part of the metamodel or EMF and Xpand/Xtend infrastructure.

Java structure metamodel is a part of Eclipse JDT framework, which can parse and describe structure of Java source code. It has not been designed as an EMF model. It is a normal set of Java classes accessed like a model by Xtend (M2M language). This is an exemplary usage of an abstract type system in Xpand and Xtend. This type system supports different meta-metamodels at the same time (e.g. EMOF and XML Schema). Here the JavaBeans metamodel is used to access Eclipse classes that represent Java code. Eclipse handles source code parsing, building the model and automatic synchronization. Created Java objects can be accessed like EMF models in Xtend transformation.

A problem with the legacy model has been identified though. Used framework is optimized for runtime speed and memory demand in mind. Some relations...
are not in the model itself and some structure information is put in specially formed strings. Functions that can handle such data and provide information about relations are present, but as outside functions. Xtend allows for definition of extensions – functions that may be defined externally, but can be used like methods on a given class. Custom extensions have been developed to simplify manipulations on that model. They decorate a model so its handling looks more object-oriented. For instance, to get a superclass of a class in a plain model, new TypeHierarchy needs to be created and a superclass should be read from it:

\[\text{type}.\text{newSupertypeHierarchy(} \text{null})\text{.getSuperclass(} \text{type})\]

In an enhanced model the same result can be achieved with:

\[\text{type}.\text{superclass(} \)\]

**Hibernate mappings model** is derived from DTD files that come with Hibernate. There is no adapter or a type system for DTD files, so they cannot be used directly as a metamodel. Transformation of DTD into XML Schema was the simplest solution. Fortunately, it is a common problem and tools that support it, exists. In this prototype a free open source tool – Trang [76] has been used.

Second problem with legacy models has been noticed during this step. XML Schema file (XSD), created in this process, is redundant and uses anonymous types. Additionally many types are not specified (no distinction between numbers and strings) or not specified explicitly (boolean is realised as an enumeration). Quality of a metamodel raises complexity of transformation, thus decision has been made for further refinement of the XSD file. Automatic transformation, written in Xtend has been created to achieve that. It assigns names to types and extracts them to minimise redundancy. It also introduces some known types. Types in Java model are not resolved, they are described as unique strings that need to be evaluated in proper context to get an actual type.

For instance, before refinement, the boolean type was represented as a restriction on the string type, that was accepting two values: “true” or “false”. Code on Listing 6.1 checks if such a situation takes place and changes it to use the standard type: “xs:boolean”.

```java
Listing 6.1: Replace true/false enumeration with “xs:boolean”.

handleBoolean(Attribute a, SchemaType schema):
    if a.hasEnumeration()
        && {"true","false"}.containsAll(a.valueSet())
    then (a.setType(createQName("xs:boolean")))
```

**Hibernate mappings model** is derived from DTD files that come with Hibernate. There is no adapter or a type system for DTD files, so they cannot be used directly as a metamodel. Transformation of DTD into XML Schema was the simplest solution. Fortunately, it is a common problem and tools that support it, exists. In this prototype a free open source tool – Trang [76] has been used.

Second problem with legacy models has been noticed during this step. XML Schema file (XSD), created in this process, is redundant and uses anonymous types. Additionally many types are not specified (no distinction between numbers and strings) or not specified explicitly (boolean is realised as an enumeration). Quality of a metamodel raises complexity of transformation, thus decision has been made for further refinement of the XSD file. Automatic transformation, written in Xtend has been created to achieve that. It assigns names to types and extracts them to minimise redundancy. It also introduces some known types. Types in Java model are not resolved, they are described as unique strings that need to be evaluated in proper context to get an actual type.

For instance, before refinement, the boolean type was represented as a restriction on the string type, that was accepting two values: “true” or “false”. Code on Listing 6.1 checks if such a situation takes place and changes it to use the standard type: “xs:boolean”.

```java
Listing 6.1: Replace true/false enumeration with “xs:boolean”.

handleBoolean(Attribute a, SchemaType schema):
    if a.hasEnumeration()
        && {"true","false"}.containsAll(a.valueSet())
    then (a.setType(createQName("xs:boolean")))
```
a. setSimpleType(null)
) else null
;

Eventually, two metamodels derived from DTD are used and transformation can be reused to derive easily adjusted to handle new DTD files.

PSM1 to PIM transformation

In this step, a Java model is transformed into UML. Output model is better formed, all relations are explicit and types are resolved. Information that is not modelled in a PSM, but is present in code, is added to UML as textual constraints. For instance, code on Listing 6.2 maps a Java method to the UML operation and adds methods body as an operation body constraint (last line).

Listing 6.2: Change Java method to UML operation.

```java
create Operation toOperation (IMethod m):
    setName (m.elementName) ->
    setVisibility (m.getVisibility()) ->
    setIsStatic (m.isStatic()) ->
    setIsAbstract (m.isAbstract()) ->
    ownedParameter.addAll (m.getParameters()) ->
    setBodyCondition (sourceConstraint ('Method', m.source))
;
```

Code generation and the (lack of) target PSM

Code generation is the last step of the final modernization workflow, but it has been developed after transformation to a PIM. Such sequence has been planned to create the identity transformation. Once code generation is created and checked, developer can follow progress done with other steps of modernization.

However there is another problem. The used Java metamodel is basically a read-only model. To introduce changes in that model, a different API needs to be used. As such, it cannot work as a target metamodel. Fortunately a PIM model used in this prototype, has enough details to generate code directly from it. It is simpler than creating a PSM2, PIM to PSM2 transformation and code generation from PSM2 – as it is suggested in the RMA workflow.

Another problem is seen here. Although many Java code generators that take UML as input, exist, they were integrated to the one UML tool, using some specific profiles, not detailed enough, used a different subset of the UML, which was hard
to customise etc. Eventually direct generation of Java code from the UML model with tags, has been created from scratch.

Changing Hibernate version

Change of version is done on an UML level. It consists of two parts.

- Namespace change
- Addition of new methods

Namespace change is realised by change in a package structure. Code in application uses import statements and there are now Hibernate-specific, fully qualified names in tags, so no textual, unsafe changes were needed.

After those changes, prototype have been abandoned. Justification for that is described letter. Following descriptions of steps contains information, how the steps had been planned. Feasibility of described step have been evaluated in design phase and confirmed, but they have not been provided.

Addition of new methods had been planned as handmade changes to the generated code that would be isolated, but this step was not developed as prototype was abandoned.

Merging with Hibernate model

Mapping Hibernate concepts from XML files, to same concepts as annotations can be done in couple of ways. Hibernate annotations can be created in an UML model. This requires to select a convention, which may need a profile for Java annotations. When convention for modeling annotations in UML is ready, it needs to be considered in the code generation. A second way is to generate annotations in the code generation phase, so no additional model representation will need to be used. The first approach can be realised in the “pull” and “push” way. In “push” transformation, mappings model is traversed and information from it is “pushed” into the UML. In “pull” transformation the UML model is traversed and additional information is “pulled” from Hibernate model to enchant UML or generate text directly. If annotations are created in code generation step, the “pull” transformation style need to be used, but in such approach PIM model is not needed.

Step 6: Verify migration through testing.

During this step the migration correctness is verified. In our case this step did not require much effort, because unit tests for the project we migrate, have been
provided. When the tests run correctly, it gives us a general outlook, that the migration has been finished successfully. But dealing with huge legacy systems, this step’s effort makes up even 45% of the whole modernization enterprise [12].

6.4 Exemplary modernization project: the DTA

*Author: Krzysztof Kowalczyk*

**Step 1: Requirements analysis**

This step is analogous to the respective step in the first prototype.

**Step 2: Development of migration tools**

a) Identification of needed migration tools: This step is analogous to the respective step in the first prototype.

b) Acquiring and integration of migration tools:

   i A metamodeling standard – Eclipse Modeling Framework
   
   ii A model discoverer – MoDisco J2SE5 and Xtend integration XML (Eclipse Xpand)
   
   iii A model manipulator – Xtend (Eclipse Xpand)
   
   iv A code generator – Xpand (Eclipse Xpand)

   c)

**Step 3: Target platform preparation**

This step is analogous to the respective step in the first prototype.

**Step 4 and 5: Definition of an automated migration; Migration execution**

In this case the migration process is based on the DTA workflow [section 5.3.1](#). Specificity of this project required introducing some changes to the workflow, nevertheless the general concept was preserved.

1. Model discovery – transformation: code → PSM1
2. PSMs transformation – transformation: PSM1 → PSM2

3. Code generation – transformation: PSM2 → code

PSM1 and PSM2 are described with the same metamodel – MoDisco J2SE5 [73].

Model discovery

The prototype 2 uses the MoDisco JavaDiscoverer with dedicated J2SE5 Java code model. This component reuses Eclipse JDT parser (the same framework, but much more detailed and complex part of compiler than it was used in the prototype one) and discovers an EMF model from the Java code.

J2SE5 metamodel is a complete, resolved Java 1.5 code metamodel (ASG-like), but semantically irrelevant information are still available (more typical for AST). For instance it contains comments and information about how the name of pointed type has been used (e.g. if it was fully qualified name). This kind of information is semantically irrelevant, but such metamodel (the resolved AST) has been considered particularly useful. It contains a full and well formed Java model. It allows to generate code that is exactly the same as the original (only formatting would be different), but it does not contains any irrelevant details. It also works well as a PSM in a PIM to PSM transformation, then Java code is created in the M2M step.
PSMs transformation

**Merging with the Hibernate model** Most value is created in this step. Hibernate mapping files are used as models in a similar way to the prototype one. For each class in the Java model, the respective mapping file is traversed, and for each element from Hibernate model, an new element in the J2SE5 application model is created. This part can be considered as a model mapping, because elements from the Hibernate metamodel are mapped to elements from the Java metamodel – J2SE5. It can be also considered as model merging, as all information from one model is merged into another. The exemplary function, written in Xtend language, shown on [Listing 6.3](#) creates a *Column* annotation.

```xtend
handleColumn (Object p, BodyDeclaration fd):
    p.mapInto(fd,"Column","column1","name") ->
    p.mapInto(fd,"Column","length") ->
    p.mapInto(fd,"Column","unique") ->
    p.mapInto(fd,"Column","notNull","! nullable") ->
    p.mapInto(fd,"Column","update","updatable") ->
    p.mapInto(fd,"Column","insert","insertable") ->
    null
```

Function on [Listing 6.3](#) reads properties from Hibernate model element (Object p) and map them into annotation values that are added to Java element. This function create annotation “Column” and its properties for instance “nullable” which is initialised with negated value of Hibernate model element property “notNull”.

Not all elements of the Hibernate metamodel are handled, because of time constraints. Nevertheless, a full transformation is not needed to proof feasibility of this prototype design.

The implementation of this step showed again that legacy models, that were not designed with MDE in mind, are not well formed. Even though the Hibernate mapping metamodel has been refactored, its class structure is redundant. Similar properties exist in number of classes and they are handled in the same way. In a good model, there would be a parent type that would extract such redundant elements. Unfortunately, such relations could not be better described in DTD, so the derived model will have lower quality than a handmade one.

Xtend used for M2M is a statically typed language, so each occurrence of the same field, but in a different class without a common ancestor, needs to be handled separately.
A static type system and a low quality model would lead to a very redundant code. Fortunately, the Xtend type system allows for the reflective programming, so dynamic typing can be simulated. Same properties in different Hibernate metamodel classes have the same names, so we can access them dynamically by name (“the duck typing”). We have used this possibility, what can be seen on Listing 6.3. Function mapInto, which takes the following parameters: `javaElement`, `annotationName`, `sourceProperty`, `targetProperty` is a dynamic function. It is a quite sophisticated function, but its usage is straightforward and can be reused whenever annotations need to be created. It maps value of a property, named “sourceProperty” in an object, into a property “targetProperty” in an annotation “annotationName”, which is attached to the “javaElement”. If such an annotation did not exist, it is created. This function allows to write one mapping that will handle each occurrence of a given property type (a conceptual type, not a modelled type).

**In-place changes**

The last step of the workflow is realised by two substeps:

- To adjust the source code to use Hibernate 3.x libraries instead of Hibernate 2.x.
- To increase the quality of the code.

**Version change** It was described earlier what is needed to change a Hibernate version. In this prototype, an M2M transformation changes package name and adds new relations (e.g. an interface implementation). This step of the workflow was realised before the previous step. Function that run those transformation is quite simple and it is shown on Listing 6.4.

Listing 6.4: Changes in model to use the Hibernate 3.x namespace and some other changes.

```java
refactor(Model this):
    handleUserType() ->
    handleCompositeUserType() ->
    findPackage("net.sf.hibernate.expression")
    .setName("criterion") ->
    mergePackage(
        findPackage("net.sf.hibernate"),
        package("org.hibernate")
    )
```
“mergePackage” invocation in function on Listing 6.4 (or in Xtend terminology: “extension”) moves all elements contained in the package: “net.sf.hibernate” to the package: “org.hibernate”. A different type of a namespace change is needed for “UserType” and “CompositeUserType”. They also need changes in the structure. Thus, they are handled before. After those changes, generated code is in an invalid state. It needs additional handmade customisation, that is described later.

Handling handmade changes Custom changes are integrated with the generated code in the last step of the workflow. It cuts a code generation process to introduce changes in this code. To implement missing methods and other changes, one needs to understand the objectives that source code is achieving.

After the step in which the version change had been made, the application was in an invalid state, so custom changes to the code were made, to fix it. Then, those changes were isolated in aspects. This allowed for an iterative development process. Those aspects cut the textual generation of Java code and add manually implemented code fragments in places where they belong.

Xpand aspects in general define what fragment of a code generation will be cut (a pointcut) and code that will be executed instead (an advice). New code checks if a place of aspect invocation, is the place where injection of manual changes is needed. If this is the case, then code contained in advice is emitted. In other case, generation proceeds as normal.

An exemplary aspect is shown on Listing 6.5. A pointcut is set to run each time, when TypeBodyHandle template is executed for AbstractTypeDeclaration – a J2SE5 model class or its subtypes. This occurs during generation of each Java type at the end of a type body. An inside advice is checked, if type implements “org.hibernate.property.Getter” interface. This interface gained a new method in Hibernate 3.x so it needs to be implemented. If check passes, an implementation of the new method is generated.

Listing 6.5: Exemplary Xpand aspect.
<<AROUND toJava::TypeBodyHandle FOR AbstractTypeDeclaration>>
<<IF isImplementing(‘org.hibernate.property.Getter’)>>
@Override
public Object getForInsert(Object target,
java.util.Map mergeMap,
org.hibernate.engine.SessionImplementor arg2
) throws HibernateException {
    return get( target );
}
Quality of code  The last M2M step was planned in order to increase the code quality. Unfortunately, introduction of generics with help of type inference has not been created. An automatic introduction of generics is essentially complex, thus time consuming. There are no technical issues left to deal with, but because time constraints, it was not developed. In this step, readability of code is increased, by organising code imports (using non qualified names where appropriate and introducing “import” statements). Hibernate/JPA annotations have different default values than Hibernate XML mapping files. After merging with an XML mapping, each annotation value is explicit, hence amount of code can be reduced by removing default values. Exemplary effect of such an action is shown on Listing 6.6.

Listing 6.6: Generated code with and without defaults values.

```java
// after model merge
@Column(name = "NAME", length = 255,
    unique = false, nullable = false,
    updatable = false, insertable = true)
private String name;

// after concerning default values
@Column(nullable = false, updatable = false)
private String name;
```

Code generation

The J2SE5 is a direct model of Java code, but there was no Java code generation available for it. A code generation is a last step of workflow, but it was developed after model discoverer was configured. This way, an identity transformation has been developed. The code generation is straightforward, as all metamodel elements maps to exactly one construction in a target code, what was shown in Listing 6.7.

Listing 6.7: Exemplary templates for Java generation.

```java
<<DEFINE Expression FOR NullLiteral>>
null
<<ENDDEFINE>>

<<DEFINE Statement FOR IfStatement>>
First template generate "null" string for each occurrences of NullLiteral type in Java model. Seconf template generate if / else statement. There are many templates with name Expression and Statement. Template that match the type (e.g. NullLiteral) in the best way is executed.

Step 6: Verify migration through testing.

This step is analogous to the respective step in the first prototype.

6.5 Feasibility study analysis

Author: Krzysztof Kowalczyk

6.5.1 Construction process

In both prototypes the following sequence of steps has been found appropriate:

- Select and prepare metamodels
- Develop identity transformation (read code model and write it back)
- Develop transformation for a version change
- Develop and isolate handmade changes
- Develop the mapping and refine code generation if needed

After completion of one step, next step is handled. Unfortunately, effort put into modernization does not provide value for developers until last step is finished. Because of low adoption of MDE and MDSM, steps before the last one are complex. Therefore big effort must be put into modernization, before any results are visible. This implicates a risk and should be addressed. Because of that, each of this steps has been developed iteratively and only in scope that was needed at the given moment (e.g. code generation could not generate annotations until they were introduced to model by the transformation in the last step).
6.5.2 Reusability of components

Finished prototype consist of new or acquired elements, that can be grouped into four classes of reusability:

- Universal MDE tools (EMF, M2M, M2T, workflow engine)
- Specific for Java(J2SE5) transformations
  - Model Discoverer (2976 ploc)
  - J2SE5 metamodel (16322 ploc)
  - J2SE5 model creation, query and reactor functions (430 loc)
  - Code generation (320 loc)
- Specific for Hibernate 2 to Hibernate 3 transformation (300 loc)
- Specific for modernization of this application (340 loc)

Where loc and ploc are defined as:

- **loc** Lines of code – lines of code, with empty lines and comments. Java, Xtend, Xpand and XML(workflow) code lines have been counted. Counted manually.

- **ploc** Perfect lines of code – lines of pure, formatted Java source code (without comments, empty lines and JavaDoc). Counted with usage of a developed tool.

It is estimated that no more than 600 hibernate specific lines of code (loc) and 400 of J2SE5 specific loc are needed to finish the merging part of workflow (estimation based on number of elements that have not been handled in model). Approximately half of developed code can be potentially reused in any Java oriented MDE project (as it can handle any Java 1.4 and 1.5 code) and additional 300 existing lines of code can be used when new, but similar problem is solved. That code should be possible to reuse without any changes, but it may need additional code, because not all elements have been mapped (Hibernate) or are generated (Java). Based on that, we claim that prototype 2 has good reusability of components.

6.5.3 Main challenge in the exemplary modernization

Each of measured values is small in comparison to the size of J2SE5 metamodel implementation. It is 16322 ploc. This is an order of magnitude higher than
size of other workflow steps. However, this code is not written by hand, but generated by EMF as an implementation of the J2SE5 metamodel, defined in Ecore. Java metamodel (J2SE5) contains: 101 classes, 150 references and 45 attributes. Inherited attributes and references are not counted. This provides almost 300 elements in metamodel. The EMF generates Java representation of this model, specially edit code (that allows to undo changes), persistence code etc. This creates a huge amount of code, hence ploc value does not exactly reflect complexity of the metamodel creation.

Loc measure values count different types of code and white lines and as such are inaccurate and can be used only for illustrative purpose. Moreover the modernization is not finished. Nevertheless, the complexity of model discovery phase is clearly visible in both prototypes. Complete Java model discoverer consist of almost 3000 pure Java lines of code. This is two times more code that has been developed, even though loc measure overestimates the size of actually written code. For the time of our study, only one complete model discoverer has been found, fortunately it was dedicated for Java. This shows, that EMF and Java are primary platforms for implementing model based solutions.
Chapter 7

Challenges and recommendations in application of MDSM

7.1 Availability of existing solutions

Author: Anna Kwiecinska

7.1.1 MDA standards useful in MDSM

The most important MDA standards that could be used in MDSM are:

- MOF as a metamodeling standard to define and manage models
- QVT as a standard for definition of M2M transformations
- MOF2Text as a standard for definition of code generation
- UML or KDM as PIM
- KDM and/or ASTM as PSM

MDA does not specify how to organise transformations in a workflow or how to transform code into model. Three technologies have been found, that can be considered as T2M tools (Xtext, EMFText, TCS). They can be used to generate model discoverers. T2M steps is not standardized, although those tools are generally similar and all of them use a EBNF-like grammar and construct a EMF-based model. MDSM partially overlaps with ADM.

7.1.2 Availability of existing solutions

Gained knowledge of tools and transformations availability has been summarised in Table 7.1
EMF usage has been assumed, as it provides the best support for each require-
ment, what is discussed in following sections. Support for CMOF is not full, but
still relatively good. On Table 7.1 we claim that metamodeling support is suf-
ficient, because EMF allows for definition of a metamodel easily in a number of
ways, as it has been shown on Figure 7.1.

As it can be seen on Table 7.2, M2M and M2T are widely available. M2T is
mature, whereas M2M is still a field of research. Standard compliance is good if we
consider the quality of the standards, but support for QVT implementation, that
provides both relations and operational mappings, is still not available.

Some PIMs defined by the OMG are available, especially UML and KDM, but
their suitability for modernization is arguable. Finalization of ASTM is needed
for better support of MDSM. Availability of PSMs is not sufficient. Language
metamodels are the most general and reusable PSMs, but they are coupled to
model discoverers, hence they are complex. More general PSMs are less reusable
and slightly more common. A consequence of problems with PSMs and early
stage of M2M languages, is that reusable transformations are not available. Lack
of transformation that could be reused and not sufficient PIMs increase complexity
of PIM based approaches.

Practical nonexistence of low level PSMs is another factor that influence com-
plexity. The first prototype was aimed to be an EPIM approach, because that
made the discovery step easier. But generation step needs much more information
to generate the behavioural part, so tags have been added to the model. Eventu-
ally, it is hard to classify a kind of approach, realised in first prototype. The
model is based on UML Class diagram part, but tags used in this model provide
full, but unreadable information. The information can be used in the code gener-

<table>
<thead>
<tr>
<th></th>
<th>Availability</th>
<th>Standard support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metamodeling</td>
<td>Sufficient</td>
<td>Partial</td>
</tr>
<tr>
<td>Model discoverers</td>
<td>Insufficient</td>
<td>Not covered</td>
</tr>
<tr>
<td>Model discoverers gen.</td>
<td>Sufficient</td>
<td>Not covered</td>
</tr>
<tr>
<td>M2M</td>
<td>Sufficient</td>
<td>Partial</td>
</tr>
<tr>
<td>M2T</td>
<td>Very good</td>
<td>Good</td>
</tr>
<tr>
<td>Existing PIMs</td>
<td>Arguable</td>
<td>Partial</td>
</tr>
<tr>
<td>Existing PSMs</td>
<td>Insufficient</td>
<td>Partial</td>
</tr>
<tr>
<td>Existing transformations</td>
<td>Insufficient</td>
<td>Not covered</td>
</tr>
</tbody>
</table>

Table 7.1: Support for MDSM tools

Table 7.2: Support for MDSM tools
ation, but it is not possible to reuse it in the M2M phase. Some changes to model can break semantic of the model. All things considered, complexity of CPIM and direct transformation approaches are severely affected by lack of detailed model discoverer. Additionally, model discovery step can be simplified in some specific situations when EPIM is used. On the other hand, the EPIM approach is not sufficient in some kinds of modernizations, what has been already discussed.

7.2 Challenges

Author: Krzysztof Kowalczyk

To conduct model driven software modernization we need to acquire and integrate a set of tools. Here, the encountered problems are described. Although, MDA provides MOF, QVT and MOF2Text standards that could be reused in MDSM, they have not been chosen, because many of those problems do not occur when non standard tools are chosen. In particular we had to deal with following problems before any approach could be applied:

- Problems with support for MOF standard
- Immature transformations standards (QVT, MOF2Text)
- Lack of existing transformations that could be easily reused
- Lack of a standard for definition of cartridges
- Weak support for customisation of transformation
- Complexity of model discovering (T2M)

Problem with metamodeling standard

Risk is connected with the MOF standard, which is recommended by OMG, but is not well supported by industry. Norminative models (EMOF, CMOF) of version 2.0 are not available in XMI, although they are needed by other standards. MOF has an official mapping to the Java language – Java Metadata Interface (JMI) [49] and it is supported natively by Metadata Repository (MDR) [71], but it supports only older MOF versions (1.3, 1.4) and XMI versions accordingly (1.1, 1.2). Nevertheless languages for model transformation and code generation that can work with MDR, are available, as it is shown in Table 7.2.

CMOF is an issue even in scope of OMG specifications. OCL does not have special version for it, thus QVT has problems with usage of CMOF models. This issue is explicitly mentioned in QVT standard [34].
<table>
<thead>
<tr>
<th>Solution</th>
<th>Metamodel defined in</th>
<th>M2M support</th>
<th>M2T support</th>
<th>T2M support</th>
<th>Supported persistence and interchange format</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML based</td>
<td>XML Schema DTD</td>
<td>XSLT</td>
<td>XQuery</td>
<td>GPL</td>
<td>Parser Generator XML</td>
</tr>
<tr>
<td>MDR</td>
<td>MOF 1.3, 1.4 Epsilon ETL</td>
<td>Java</td>
<td>Epsilon EGL</td>
<td>Not found</td>
<td>XMI 1.1, 1.2</td>
</tr>
<tr>
<td>EMF</td>
<td>Ecore (preferred, EMOF like)</td>
<td>UML (class diagram subset)</td>
<td>MOF2Text (e.g. Acceleo)</td>
<td>JET</td>
<td>XM Textual notations (with T2M)</td>
</tr>
<tr>
<td></td>
<td>XML Schema</td>
<td>Xpand/Xtend</td>
<td>Viatra</td>
<td>MOFScript</td>
<td>HUTN RDBMS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>UMLX</td>
<td>others</td>
<td></td>
</tr>
<tr>
<td>KM3</td>
<td>(textual DSL)</td>
<td>ATL</td>
<td>Not found</td>
<td>TCS</td>
<td>Textual notations (with TCS)</td>
</tr>
</tbody>
</table>

Table 7.2: Different solution toward metamodeling.
As it is pointed in [26], tools that support XMI before version 2.0, produce XMI files that are incompatible between tools. As there are neither standard APIs, nor reference implementations of new versions of MOF, there are problems with support for XMI, because XMI file need a metamodel definition. Metamodels were described in proprietary technologies by tools vendors, what probably had influence on weak interoperability of XMI models, especially interoperability between UML tools had been poor [26] as there are many incompatibilities between XMI dialects [42].

Summary of the problems with metamodeling:

- MOF is not well supported
- CMOF provides number of interoperability issues with other standards
- Problems of XMI interoperability between tools when different metamodel implementations are used

7.2.1 Problem with complexity of model discovery

Model discovery is the first and obligatory step of MDSM. Reading content of well-formed artifacts with dedicated manipulation API like XML files or EMF models is easy, but most of existing applications have been created in programming languages. Although each programming language needs a kind of compilator or interpreter, their API is not necessarily available. Still, code model need to be exposed to conduct modernization. Unfortunately there is a huge number of computer languages and not many model discoverers. This is an important issue as creating code parser is a task that needs a great effort (because of the complexity of parsing itself and programming languages complexity) that can exceeds effort of the rest of modernization steps. In prototype 2, the discoverer used an existing parser and still its complexity exceeds complexity of created modernization. Availability of existing parsers depends on language. Ideally, parser would provide the access to a simple ASG representation of code. Unfortunately, parsers have not been build with MDE in mind, so API can be not available or the used structure can be suboptimal (as it was the case in prototype 1). If parser is created in a language “X”, the layer that constructs the PSM will need to be provided in language compatible with that language.

In short:

- Existing model discoverers are rare
- Model discovery is a complex process and creating one from scratch, needs huge effort
Problem of immaturity of QVT and MOF2Text

Although there is a number of projects that implement QVT in some scope (ModelMorf, SmartQVT, MediniQVT, Eclipse M2M, UMLX), none of them supports the full standard. Additionally, we have not found any implementation, which would allow for using both: QVT Relational (QVTR) and QVT Operational Mappings (QVTO), although the standard [34] defines how to combine them, to create hybrid transformations. QVTR is less verbose than QVTO on simple examples (what can be seen on examples provided with the standard [34]). QVTR has also a very good support for tracing and allows to define bidirectional transformations. With this in mind, QVTO is still useful, as some transformations are difficult (or even impossible) to realise declaratively [18]. Verbosity of imperative approach on simple examples, weakness of declarative approach and a need for tracing are recurring issues of M2M languages. Another problem is an integration with MOF2Text. Rules how QVT and MOF2Text can be integrated are not specified by standards. As for the time of writing there is no support for using QVT side effect free queries (“Functions” from QVT Base package [34]) in MOF2Text templates, even though they are common for both specifications.

Ambiguous standards are source of issues in standard implementations. List of known issues concerning QVT standard, can be found on OMG website [74], as for 16/08/2009 there are 95 not resolved issues. Some of them are critical. Hence, it is impossible to provide functional implementation and conform to standard at the same time. This leads to incompatibilities between implementations. Obeo, the company that develops implementations of declarative QVT [65] and MOF2Text [58] for Eclipse project, has pointed following problems with these standards during Eclipse/OMG symposium in 2008 [75]:

- Differences in standard libraries
- Not clear semantic of some concepts
- Naming inconsistency and braking of conventions
- Redundancy
- Serious lacks in functionality

In context of MDSM it is important to have possibility to reuse existing frameworks and models. This is hard as legacy systems does not use modeling standards as, in most cases, legacy systems where created when modelling standards where not available or not mature. OMG standards expect MOF-compliant models and QVT/MOF2Text implementations expect EMF-compliant models. Thus code parsing need as well as model construction need to be used. After analysis of other model transformation techniques, we have discovered that M2M and M2T
components can provide better support for integration with legacy models. For example Epsilon languages use Epsilon Model Connectivity Layer (EMC), a solution that allow using all languages with different meta metamodels. Components that provide ability to handle new meta metamodel type are called “drivers”. Epsilon provides drivers for EMF, MDR and others. Eclipse M2T Xpand language family likewise provide a uniform abstraction layer over different meta metamodels. A component that provides support for additional meta metamodel is called “adapter”. Xpand has defined adapters for EMF, Eclipse UML, Java Beans and XML Schema. XML Schema adapter makes XML files accessible to Xpand languages. This is valuable, because XML is widely used. Java Beans adapter allows to use existing Java classes as models. This allows to use existing frameworks, in particular parsers, without a need to provide an additional integration layer.

In summary we have faced following challenges with transformation technologies:

- Ambiguous and inconsistent QVT and MOF2Text standards
- No integration between QVT and MOF2Text in existing implementations
- No integration between QVTO and QVTR in existing implementations
- Verbosity of QVTO
- No support for many meta metamodels in QVT and MOF2Text

### Problem with reusing transformations and metamodels

We did not find cartridges that we could reuse. Cartridges we have found support for instance:

- Generation of classes (usually with persistence capabilities) from UML class diagrams
- Scaffolding of user interface based on class diagrams

Those cartridges (e.g. Fornax-Platform, AndroMDA) are not using PIM/PSM division. They generate code directly from UML (also from textual in Fornax) models. We could consider models appropriate for those cartridges as PSM, but developers are encouraged to manipulate those models directly. Additionally they are not detailed enough to be useful in our context. Another problem can be noticed here. Cartridges are not available in standard solutions.

AndroMDA is one of the most popular MDA-like code generators, but it uses template engine, that is not intended for M2T. Special layer for integration between models and templates is used (AndroMDA metafacades). Moreover, there
is no standardization of cartridges or suggestions how to change model transformations. It is not clear, how cartridges could be reused or their behavior altered. Different projects use different solutions. For example, Epsilon uses popular Java build tool ANT, to describe workflow. Xpand uses dedicated, extensible framework for defining workflows – Eclipse Modeling Workflow Engine (MWE). Non standard languages provide different additions that support cartridge Xpand language family uses Aspect-Oriented Programming (AOP) paradigm to allow easy cartridge adoption (as well as tracing and others). In Xpand it is possible to intercept any M2M function or M2T template invocation. Developer needs to create a mask to select functions to be intercepted, and define what should be done instead. In comparison, QVT transformation can transform another QVT transformation. Although it is powerful, we found it complex. Moreover we have found that support for this QVT feature is weak.

We also had a problem in obtaining PSM metamodels. Platforms and frameworks vendors do not create such models. They are created by additional parties, for very general platforms like J2SE5 in MoDisco [73], or very specific like those available in Fornax-Platform [67]. There is a huge number of frameworks, which are rapidly evolving. Hence there is small possibility that PSM metamodels will be available for our specific framework. Nevertheless even general metamodels are rare. We have found only few metamodels for Java and C# languages. Creating proper PSM metamodel is hard (even when existing tools are reused, as in J2SE5 discoverer) and there exist a risk that created metamodel will be on improper level of abstraction. Eventually, metamodel can be too abstract, so models will miss important information. It can be defined also at too low level, so the effort when using the model is greater.

In summary we faced following challenges with cartridges and PSMs:

- Definition of cartridges is not standardized
- Support for transformation refinement could be better in standard transformation languages
- Low availability of cartridges that could be reused
- Low availability of existing Platform Specific Models

7.3 Solutions

*Author: Krzysztof Kowalczyk*
Select industry standard metamodeling framework

In given modernization example we have chosen Eclipse Modeling Framework as metamodeling standard and Ecore as a meta metamodel. The reason for this choice was the fact, that the Eclipse Modeling Framework (EMF) is a modeling framework with probably the highest adoption in industry. For instance, we have found 10 QVT implementations or QVT-like languages, 8 of them were EMF based, one of remaining two has been discontinued. Only one project that implement MOF2Text standard – Acceleo have been found. It is EMF-based. This leads to a conclusion that most of existing modeling projects are build on, or supports the EMF. [26] suggests that the EMF’s implementation of the UML 2.x metamodel has an influence on better interoperability of tools supporting XMI 2.0 and higher. As long as models are used on one modeling platform, the XMI files are fully interchangeable across tools based on that platform. UML tools like IBM Rational Software Architect, Borland Together, Topcased and others are using EMF UML2 metamodel natively. Other tools like MagicDraw, Visual Paradigm and others, have dedicated support for EMF UML2 import and export.

![EMF metamodels interoperability](image)

EMF was intended to be a MOF implementation, but MOF was considered too complex [40] and EMF evolved in a different direction. Eventually Ecore inspired division of MOF into CMOF and EMOF [10]. The EMF uses Ecore specific XMI
as a default format of model serialization. Nevertheless it can transparently serialize and deserialize EMOF models. CMOF is partially supported. Some models available on OMG website are defined with EMF. E.g. QVT metamodel files are available in an Ecore format (including Emof.ecore and EssentialOCL.ecore, for which there are no norminative XMI files available) and SMM beta 1 metamodel is defined as an XMI/EMOF format with Ecore specific annotations.

EMF defines mapping between UML, XML Schema and Java, it allows for defining models in any of these technologies as shown on Figure 7.1. This probably supported early adoption. EMF can generate static API for a given metamodel, provide a dynamic API for editing any EMF compliant model, support model persistence to XML files, XMI and relational databases. EMF supports distributed, concurrent model edition. EMF implementation of some standard metamodels (e.g. UML 2.2 [72], SPEM, KDM, ASTM and others) are available.

In summary, EMF is an industry standard framework with many dedicated tools for most steps of MDE. Additionally it has good support for EMOF XMI and some support for CMOF XMI, what support interoperability. Additionally, some predefined EMF-based metamodels are industry standard implementations.

Selection of mature and complete MDE toolkit

As it was mentioned earlier, there is relatively high number of available transformation tools. One can choose between many code generators and model transformation techniques. Nevertheless there is a small number of projects that aim in providing complete solutions. Complete solution should allows for M2M and M2T transformations. Those should be integrated, so one does not need to create the same query in 3 different languages. This problem with most of existing tools is described in [25]. Additional support for legacy models is desirable and can be provided. Similarly, complete solution should have support for definition of cartridges in some extendable form. Maturity of tools is also important. As most of mentioned tools are open source, developer community of tool should be investigated. It is important to protect investment in technology. Projects have different risk, concerning probability of braking changes, probability of abandoning project, or chances of new features. For instance QVT is not yet mature standard and changes in it are needed, additionally each implementation developed different workarounds for standard issues. Thus, a QVT implementation has higher probability of braking changes, than mature language like ATL[64]. Additionally, using a proprietary solution can be concerned a risk. This is caused by a huge availability of mature, open source tools.

With such requirements in mind, we can recommend 2 toolkits that allow for
solving mentioned problems: Eclipse Epsilon and Eclipse M2T Xpand. Good integration between transformation languages is a most important feature of those toolsets.

Epsilon provides:

- Domain specific, integrated languages for M2M (ETL), M2T (EGL) and validation (EVL)
- Additional special purpose languages, for instance model comparison and merging
- Support for many meta metamodels / legacy models (EMF, MDR, Z, XML, custom)
- Definition of cartridges with ANT
- Very good integration with EMF and Graphical Modelling Framework (GMF, a toolkit for graphical domain specific languages)
- Very good IDE support
- Good integration with Java
- Very good documentation and community support inside Eclipse Foundation

Xpand provides:

- Domain specific, integrated languages for M2M (Xtend), M2T(Xpand), validation (Check) and T2M (Xtext)
- Dedicated component for cartridges definition (MWE)
- Simple Aspect-Oriented Programming
- Support for many meta metamodels / legacy models (EMF, UML2 profiles, XML, Java Beans, custom)
- Good integration with EMF and GMF
- Tools to import some vendor specific UML models are available
- Very good IDE support
- Good integration with Java
- Good community support inside Eclipse Foundation
- Commercial support available

Good support for handling XML files and Java models are valuable in our context. We have chosen Xpand over Epsilon because of relatively simple and concise syntax. Xtend is imperative M2M language, with syntax that mix simplified OCL and Java. Learning curve depends on already known languages and programming paradigms. As imperative language influenced by Java should be easy to learn for Java/C# developers. It has weak support for tracing in comparison to e.g.
QVTR, what is typical for imperative languages. Nevertheless it is relatively easy to provide tracing for Xtend transformation, as dedicated functionality exists.

Fornax-Platform [67] provide set of cartridges that are based on Xpand, Xtend, Xtext and MWE. This project proves that workflow engine and AOP can be successfully used to customise and manage existing cartridges. It also provides some basic cartridges, metamodels and UML profiles. Those can be potentially reused.

In summary, selected toolkit support full scope of MDA and MDSM workflow and extend even further with support for T2M, many meta metamodels and cartridge definition. Rich functionality and high integration comes at the price of no compliance with standards. Nevertheless, as it use industry standard modeling platform, it can be integrated with standard-compliant solutions easily (EMOF/UML XMI models can used).

### Use tools supporting model discovery

In second prototype PSM on language level and additional metamodels for Hibernate mapping files have been used. Unfortunately, availability of discoverers like the used one is low. Thus, we point out some tools that can ease model discovery effort.

In general, there are a few kinds of tools, that can support T2M phase in different scope. They are shown in Table 7.3. At the beginning of study we did not find any tool that would have fit our needs. Hence integration with existing Java parser have been provided. To provide this integration, Xtend abstract type system had been used and objects generated by parser had been used directly in M2M transformation. This approach had a number of weaknesses and could be only used in very specific case. However in the beginning of year 2009, MoDisco project provided working version of J2SE5 Model Discoverer that perfectly fit our needs. Because of that tool we changed our strategy to T2M phase and different

<table>
<thead>
<tr>
<th>Tool kind</th>
<th>CST</th>
<th>AST</th>
<th>ASG</th>
<th>Model</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parser generator</td>
<td>Yes</td>
<td>No*</td>
<td>No</td>
<td>No</td>
<td>Yacc, ANTLR</td>
</tr>
<tr>
<td>Existing parser</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>GNU GCC, Eclipse JDT</td>
</tr>
<tr>
<td>T2M tool</td>
<td>Yes</td>
<td>Yes*</td>
<td>Yes</td>
<td>Yes</td>
<td>Xtext, EMFText</td>
</tr>
<tr>
<td>Model discoverer</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>MoDisco J2SE5</td>
</tr>
</tbody>
</table>

Table 7.3: Tools support for model discovery.
Table 7.3 shows selected tool kinds that can be used to discover a model from code. Model discoverer is a tool that has been designed to read code and to construct one type of models. Such projects are build for concrete programming language, database, framework, etc. They are specialised for particular toolset. For instance MoDisco J2SE5 model discoverer creates EMF Java 5 model.

If model discoverer is not available, one could consider a T2M tool. Such tool is similar to a parser generator (it aim is usually to provide support for creation of model based DSL). Textual modelling tool has better support for handling AST and name resolving (ASG) than parser generators.

On Table 7.3 support for AST and ASG is marked, but they are not used by user. Textual modeling toolkit needs a EBNF-like grammar of language to generate parser, metamodel and basic IDE support. IDE support is not needed in our use case. Sometimes the metamodel needs to be explicitly provided (e.g. in TCS[55]). Relatively small number of existing grammars for T2M tools are available, as this kind of tool are newer than classic parser generators. Nevertheless, Java and C# grammars are available in EMFText Zoo. Some DSL toolkits (e.g. TCS) need creation of mapping rules between CST and model. Xtext[57] is an example of textual T2M tool that construct EMF models and automatically handle generation of metamodel, as well as basic name resolving (no need for mapping rules as in TCS).

If grammar for a language is not available, it can be easier to reuse existing parser than writing language grammar and use generators. There are many open source compilers and advanced IDEs(with language parser's) for wide range of languages. Some of them can be used to transform code even to semantic graph level. Then GPL can be used to construct a model.

MoDisco Java discoverer is built by as extension to Eclipse JDT Java parser. A visitor is used to traverse Java AST and “bindings” (structure that correspond to ASG) to build a Java model. Unfortunately only some compilers have design that allow easy integration. In the case of Eclipse JDT compiler it is even possible to integrate it with transformation languages directly. For instance, usage of JavaBeans metamodel allow to use Eclipse JDT in Xpand/Xtend as it was done in prototype one. This show the usefulness of abstract type system used in Epsilon and Xpand.

Last kind of a tool that is worth mentioning is a parser generator (“compiler-compiler”). Parser generators greatly support construction of parsers, but user needs to provide “actions” that construct model (or AST and create model in additional step). Although in Table 7.3 no support for AST is shown, there are exceptions. ANTLR is a tool that gained great adoption in industry (e.g. Hiber-
nate use it) and allow for automatic construction of AST classes. Additionally it use “rewrite rules” that is a kind of CST to AST transformation language. It also provide big collection of language grammars. Official implementation of Java Virtual Machine also provide Java grammar in ANTLR format. Another valuable feature of ANTLR is ability to generate parser in many languages: Java, C#, C, Ruby, JavaScript and others.

7.4 Recommendations

7.4.1 General recommendations

As for today, based on practical experience as well as study of standards, literature and present case studies, some general guidelines can be stated:

- To access huge number of existing, mature and free of charge tools, as well as to protect investment in technology, select EMF as the core of modeling infrastructure. It has stable support, huge adoption and best availability of tools.

- To increase reusability of components and transformations use direct language metamodels (like J2SE5) as low level PSMs.

- Standard compliance of transformation languages is of little importance, as long as they can operate on standard compliant (preferably EMF) model. Hence, when selecting MDE toolkit, main focus should be put on its functionality and ease of adoption.

- Tight integration between different purpose MDE related languages increase productivity, but not all tools provide it.

- Language support of abstract type system allow easier adaptation of legacy models and relax dependency on used metamodeling platform.

- Support for workflow definition is needed, thus should be provided by chosen MDE toolkit.

- Although legacy metamodels can be reused, their quality can be significantly lower (as it was shown along first prototype description).

7.4.2 Recommended tools

Each of recommended tool is an open source, free for commercial use project. A selection of MDE-related tools, developed under Eclipse foundation is included in
In general we recommend usage of EMF and Xpand, Xtend, Xtext, MWE (this selection is called as “openArchitectureWare”) or Epsilon as justified in section 7.3. We provide short description of other tools that may become important in future or are appropriate for specific requirements.

**Standards implementations**

**Eclipse M2T Acceleo** [58] is the only implementation of M2T that aim at compliance with MOF2Text standard. It use experience gained with proprietry implementation of older Acceleo project. It use EMF models. Older version of Acceleo is a commercial project and is less standard-compliant.

**Eclipse M2M QVTO** [46] Eclipse has quite mature, yet still incompleate implementation of QVT Operational Mappings. Eclipse implementation (called also Operational QVT) is not fully standard compliant, for instance it use its own mechanism in place of QVT “keys”. GMF version of Xpand [68] use QVTO as a query language, hence integration with M2T technology is provided. Core and Relations languages are also developed (Eclipse M2M QVTR, called also Declarative QVT), but at the time of writing, QVTR has not been released. Each of those projects use EMF models.

**MediniQVT** [50] We considered mediniQVT as best available implementation of QVT Relations at the time of writing. It use EMF models. Although, the engine is free, MediniQVT IDE tools (e.g. QVT Relations language editor) are not free for commercial use.

**Non standard tools**

**ATL** [64] is one of the oldest and the most mature model-to-model transformation language. It is a hybrid language that is based on OCL. Some existing projects use ATL for M2M transformation (for instance MoDisco). ATL Zoo [41] provides many exemplary transformations developed in ATL. It is well integrated with EMF, but different metamodels can also be used.
Chapter 8

Conclusions

Authors: Krzysztof Kowalczyk, Anna Kwiecińska

8.1 Summary of the research

In this thesis we have described the problem of legacy code and the need for its modernization. In response to the market needs, a concept of Model-Driven Software Modernization has been described and demonstrated as one of possible solutions to this problem. Moreover, two MDSM-based approaches have been presented. One of them (the “Reverse” MDA approach) is strongly related to MDA standards and has already been in use in industry, the second approach (the Direct Transformation Approach) is our proposition as potentially simpler approach for a specific type of problems.

Qualitative analysis justifies, why the proposed approach can be simpler than RMA. Additionally, a real world exemplary modernization project proves feasibility of the DTA.

MDSM is supported by a number of MDA standards, mainly: MOF, QVT, MOF2Text and ADM-related standards like KDM and ASTM. But most of those standards have been identified as not yet efficient in practice. They are not well integrated, incompatible between implementations or functionally incomplete. Text-to-model transformations and workflow definition is not standardised. Finally, we recommend to evaluate also non standard solutions while selecting MDE tools. Some guidelines toward tools selection have been outlined in chapter 7. In particular we have pointed out the toolsets which allow to put MDSM (or MDE techniques in general) into practice in an easy and effective way.

To summarise, we have presented the idea of MDSM, proposed different approaches toward MDSM for a subset of modernization problems and we have shown
their feasibility by solving a real modernization problem. Hence, the aim of the thesis has been achieved.

8.2 High cost of entry

We have identified that cost of introducing MDSM is very high. In the best scenario, only model-to-model transformation needs to be created and other tools can be acquired. But as for today in most cases one would have to develop a new model discoverer, design a metamodel for source and target platforms and create code generation for source platform. All of that before developing the actual modernization (before providing any added value). The direct transformation approach has been suggested to lower the cost of entry into MDSM, but still knowledge of many tools is needed.

To use MDSM one needs to understand MDE concepts and select proper tools. The second prototype have been developed in two weeks time, but it was preceded by a year and half of MDE research. We investigated MDA standards, MDA standards implementation and non standard MDE tools. We gained had to understand differences between similar standards (QVT Operational Mappings language, QVT Relations, QVT Core, KDM and ASTM) and their different implementations (e.g. we have investigated 8 implementations of QVT and each time some issues occured). We learned non standard solutions and have some knowledge about their industry adoption. We conducted experiments have examine how tools, that does not comply to MDA standards, can be used in this context. In chapter 7 we describe selected tools, that should be taken into account when a reader would like to construct an MDE or MDSM solution. It gathers projects that are mature, productive, easy to learn, and have highest probability, that they will not be abandoned (as it happened for instance with OptimalJ). We believe that, those guidelines should lower the cost of entry into MDE/MDSM field significantly.

8.3 Answers to the research questions

8.3.1 What approaches toward software modernization, based on model transformations exist today?

We have identified two approaches toward MDSM and they are described in chapter 5. ADM initiative standardised an approach toward software modernization, which is similar to the RMA approach, identified in this thesis. This approach has also been successfully applied in industry [12]. Sodifrance created its own tools
(M2M, M2T, PIM, model discovery) for this purpose. The prototype 2 (described in section 6.3) demonstrates that available open source tools support these steps. Thus, development of proprietary M2M and M2T tools is no longer necessary. Unfortunately, vision of reusable transformations and metamodels is not fulfilled yet. All things considered, we suggest that some kind of modernization problems can be developed in a more efficient, non standard way. The direct transformation approach toward MDSM has been described and its feasibility has been demonstrated on an example.

8.3.2 How can existing approaches toward MDSM be refined?

Answer to this research question is mostly gathered in Chapter 5. We identified that existing approach is focused on modernization with usage of PIM. This can increase complexity of modernization in some specific, but common cases. Thus, we proposed a complementary approach that is simpler to develop in some specific cases, but is not better in general.

8.3.3 What kind of tools are necessary to conduct MDSM?

Tools that are required to conduct MDSM are listed in section 5.4. Discussion of those tools from MDE perspective is provided in section 3.2 and 7.1.1. Problems that may occur with those tools are presented in chapter 7. We also proposed solutions to identified problems.

8.3.4 In what scope MDA standards and related tools support software modernization?

General MDA standards like XMI [28], MOF [29], QVT [34], MOF2Text [30] address almost all needs of the MDSM, identified in section 5.4. The MDA does not address the first step of the MDSM workflow - the M2T transformation.

As for today the main problem with MDSM is the lack of model discoverers for various platforms. Model discovery is a complex step needed by every MDSM approach, so lack of it is a serious issue.

Maturity of MDA standards is also an important issue. We have pointed out that the most general MDE components (M2M, M2T, T2M, modeling platform, KDM, UML) are already available for free, but the mature implementations do not comply with the MDA standards. Nevertheless, it is not a problem in practice, because the most mature tools and standards implementations are available and
built around one technology (EMF [40]), what makes them more interoperable. It is worth to notice that:

- Mature and productive modeling platform, tools for M2M, M2T, T2M and workflow definitions are available, but they do not comply with standards.

- UML2 and some other EMF-based metamodels are already mature and widely used.

- EMF based implementation of QVT, MOF2Text and others OMG standards emerge.

- Many (probably most) of free and mature MDE-related tools are implemented in Java and they use EMF.

- There is still small adoption of the MDA vision, hence it is hard to find reusable metamodels or transformations.

- There is no standard for T2M, but tools supporting it are available.

- There is a lack of standard for definition of cartridges and managing transformation workflow.

**Is full automation of modernization possible and cost-effective?**

During the development process of the prototype modernizations, it was identified that some changes needed by modernization, cannot be automated in general. Thus, there is no possibility to create a general, fully automated transformation, which would not need any handmade changes to code. Those changes can be done after or along with an automatic modernization, and in most cases are necessary.

**When the MDSM approach is suitable for a project?**

The MDSM is always worth considering, when the existing MDSM-based solution for a given (or similar) problem is available. If there is no existing automated modernization (what will probably be the case in most of the situations), there are some characteristics that make a project more suitable for MDSM:

- Huge usage of a small number of concepts, or recurring modernization problem (e.g. big project or many projects with similar structure, usage of same libraries etc. – cost of entry in MDSM is big, but automated approach is easily scalable)

- Availability of model discoverer (this significantly lowers the cost of MDSM, because model discoverers are complex)
• Availability of automatic tests that can be reused (testing take much effort in modernization, additionally automatic tests greatly ease development of main transformation)

• Declarative style of code (declarative code is easier to transform and generate)

8.4 Threats to validity

We have proven feasibility of proposed approach based on implementation of an exemplary modernization for a real industry modernization problem. The example represents only one kind of modernization problems, hence we cannot claim that proposed approach is feasible for all types of problems. Moreover, implemented modernization is not finished, but work that is needed to finish it, does not include new kinds of problem.

Proposed modernization is an example and it has not been conducted in a real industrial context. Application of the proposed approach in industry would increase value of this thesis, but we did not manage to cooperate with any company.

The thesis also lacks in quantitative comparison of the proposed approaches, due to the fact that we have not been able to finish the RMA prototype. We estimated some measures, but eventually we have realised that they use to many approximated values and would not be valuable.

Although we did not manage to finish RMA prototype, it is possible, that other research teams, or a company will use this approach. Based on a qualitative analysis, we think that such implementation would need more effort than DTA in general. Effort needed for concrete project depends on available tools and transformations.

8.5 Future work

8.5.1 Differences between M2M languages

As the complexity of MDSM depends on complexity of model-to-model transformation it is important that the transformation language is mature. There is high number of model transformation languages that represent variety of design decisions. In [10] it was pointed that there is not much experience in using M2M languages and it is not known what they should implement.

There are many mature and open implementations available and some projects adopts them. As it was mentioned, QVT standard cannot be considered mature.
With all this in mind, detailed evaluation of languages and features that they implement would be valuable.

It is usually hard to compare code written in different languages, but Haldstead measures (e.g. volum, length, effort)\footnote{20} may prove effective in such task. It is important to investigate how different design decisions influence complexity of a model transformation language in practice. Survey of model transformation approaches is available\cite{10}, but effects of language features on their usefulness is not discussed. Research could propose changes to standards.

8.5.2 Standard-based modernization

In developed example KDM have not been used, as it was considered an to complex (because of its general nature). In the same time, it is not detailed enough and ASTM is not finished.

Thesis writers remain sceptical on suitability of KDM as a model for PIM-based modernizations. In particular they think that components using KDM would have low reusability. Additionally it would not be easy to switch a model discoverer or transformation to different PSM. It have been noticed that in case of UML, reusability of transformations is small. In prototype one, it was noticed that simple task of generating Java from UML can be handled in many different ways. Eventually, it was considered easier to create new transformation than reuse existing one.

Research that discuss (and possibly prove) KDM usability and discuss maturity of standards important for MDSM (QVT, MOF2Text, KDM, ASTM) is needed.
Chapter 9

Appendix


Complete development environment that allow to reproduce those steps, with additional documentation is included on CD attached to the thesis. The environment is configured for running on Windows operating system with Java 1.5 installed. It also contains source code of some used tools, source code of exemplary application and one working modernization (second prototype).
Bibliography


[46] http://www.eclipse.org/m2m


[50] http://projects.ikv.de/qvt


[56] http://www.w3.org/TR/xpath


[58] http://wiki.eclipse.org/Acceleo


[64] http://www.eclipse.org/m2m/atl/
[77] http://www.eclipse.org/modeling/m2t/?project=xpand